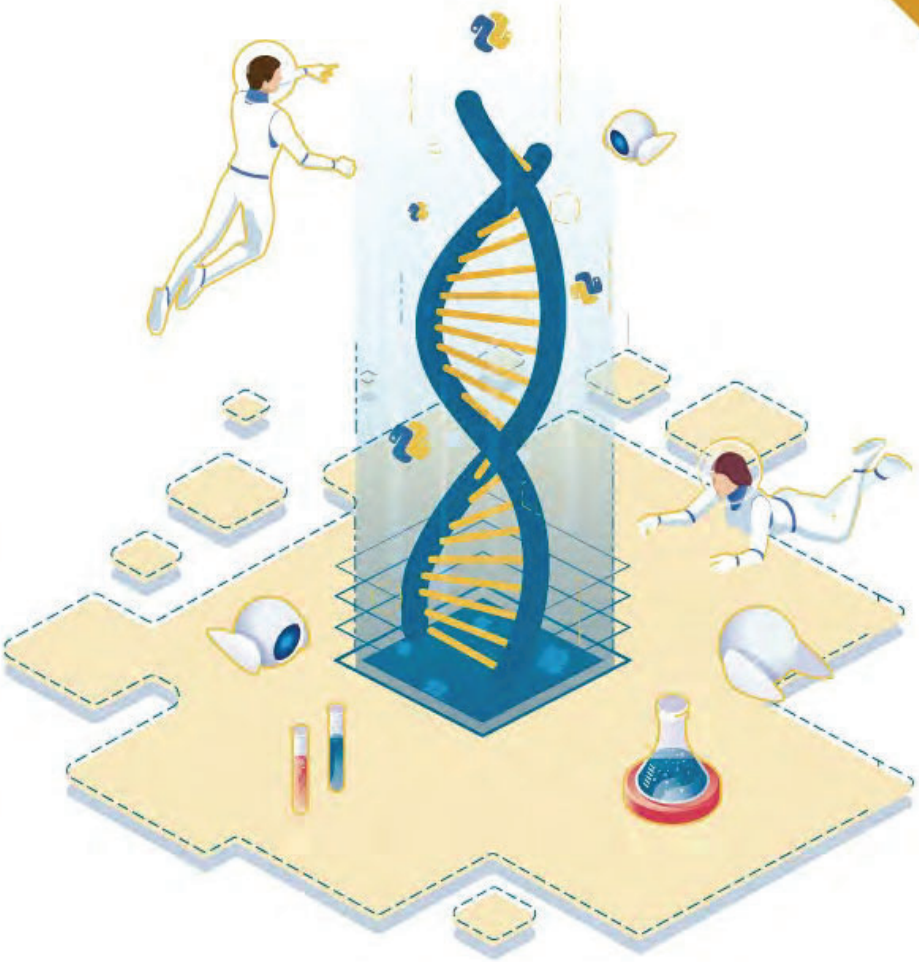
Computação evolucionária

Aplique os algoritmos genéticos com Python e Numpy



Casa do Código

EDUARDO PEREIRA

ISBN

Impresso e PDF: 978-65-86110-34-0

EPUB: 978-65-86110-36-4

MOBI: 978-65-86110-35-7

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

AGRADECIMENTOS

Quero agradecer à minha esposa Regla e às minhas filhas Victória e Kênia, pelo apoio e carinho, pois, sem isso, não teria sido possível escrever este livro. Deixo aqui meu agradecimento ao Dr. Haroldo F. de Campos Velho, pois a ideia deste livro surgiu quando participei como ouvinte da disciplina de Problemas Inversos. Agradeço também ao Jefferson Gonçalves de Souza pela revisão técnica do livro. Por fim, quero agradecer profundamente à equipe da Casa do Código, em especial à Vivian Matsui, que acompanhou de perto o desenvolvimento do livro, e ao Adriano Almeida.

SOBRE O AUTOR

Eduardo S. Pereira é mestre e doutor em Astrofísica, pelo Instituto Nacional de Pesquisas Espaciais (INPE). Possui graduação em Física pela Universidade Federal de São João Del-Rei. Também realizou pós-doutorado em Astrofísica, pelo INPE, e em Astronomia Observacional/Computacional, pela Universidade de São Paulo (USP).

Ele atua principalmente nos seguintes temas de pesquisa: Processamento de imagens e Visão Computacional, Inteligência Artificial, Cosmologia e Astrofísica Computacional. Atualmente, é professor do curso de Ciência da Computação da Faculdade Anhanguera, e desenvolve pesquisas na área de visão computacional no Laboratório de Modelagem e Estudos de Recursos Renováveis de Energia (LABREN), no Centro de Ciência do Sistema Terrestre (CCST) do INPE. Trabalha com Python há mais de 10 anos.

SOBRE O LIVRO

A computação ubíqua é uma realidade, principalmente após a popularização de smartphones. O poder computacional de um dispositivo móvel é comparável ao de um computador pessoal. Esse fato aninhado com disseminação de uso de redes sociais ajudou a alavancar o uso cotidiano da Inteligência Artificial (IA), a qual está presente nos sistemas de tradução, assistentes pessoais, propagandas customizadas, exames médicos, controle de tráfego aéreo, sites de reservas e muitos outros sistemas. Mesmo que de forma indireta, sofremos a influência de algum sistema de computacional inteligente.

Contudo, o termo IA tem um caráter amplo, podendo ser dividido em quatro grandes áreas: a de criar sistemas que agem como humanos, pensam como humanos, agem racionalmente ou pensam racionalmente. Como exemplo de cada área podemos citar: agir como humanos - serviços de atendimento ao cliente que simulam conversas; agir racionalmente - robôs que limpam a casa sem intervenção humana; pensar racionalmente - sistemas complexos que usam banco de dados e lógica para tomar decisões; pensar como humano - ainda uma ficção, como os cérebros positrônicos dos contos de Isaac Asimov.

De forma geral, os algoritmos de IA são baseados em meta-heurísticas. O termo "meta" se refere a algo que está além, ou seja, aquilo que extrapola. Já "heurística" se refere a processos cognitivos usados para tomar decisões racionais. Como exemplo de heurística, temos o ato de refazer todos os passos anteriores para procurar algo que perdemos. Nesse contexto, redes neurais

artificias, que funcionam baseadas em sistemas biológicos podem ser classificadas como meta-heurísticas, nas quais se busca criar um sistema capaz de pensar como humano.

Uma outra linha de meta-heurística de IA são aquelas baseadas no processo de evolução das espécies, atribuídas a Charles Darwin. Esse será o foco do livro, apresentar a criação de um framework computacional para a criação de sistemas inteligentes baseados em computação evolucionária. Neste livro serão apresentadas as bases teóricas do chamado algoritmo genético, em que se coloca populações de indivíduos para evoluir em busca de se resolver um problema. Uma das aplicações mais fascinantes da atualidade da computação evolucionária é o da criação de programas capazes de criar programas, a *programação genética*. Outras aplicações relevantes são determinação de melhor rota de veículos, design de circuitos, classificação de clientes, alocação de espaço físico e determinação automática de estruturas de redes neurais artificiais. Com esse último caso, vemos que existe uma interoperabilidade entre os diversos ramos da IA. Isso nos indica a relevância de se aprofundar na computação evolucionária.

Ao longo do livro alinharemos teoria e prática. Mesmo na primeira parte, em que se apresentam os fundamentos de Algoritmos Genéticos, veremos como aplicar de forma profunda fundamentos de Programação Orientada a Objetos e a utilização da biblioteca Numpy como uma extensão da linguagem Python para operar com vetores e matrizes. Também veremos como aproveitar a biblioteca Matplotlib para a manipulação de gráficos 3D e geração de animações com os dados gerados. Na segunda parte do livro, será apresentada uma aplicação prática de algoritmo genético, na solução de labirintos. Buscamos casar conceitos

fundamentais de teoria da Ciência da Computação com o processo de adaptar problemas para serem resolvidos com o que teremos apresentado nos capítulos.

PRÉ-REQUISITOS E PÚBLICO-ALVO

Este livro foi escrito para quem já possui conhecimento de programação na linguagem Python e entende os elementos básicos de Programação Orientada a Objetos, mas que procura avançar seus conhecimentos na área de Inteligência Artificial e Ciência de Dados.

Ao longo do livro, teremos tanto prática como teoria, o que permite que o desenvolvedor ou a desenvolvedora possa contextualizar seu conhecimento prévio aos novos conceitos abordados na obra para acompanhá-la com mais proveito.

Os exemplos e códigos foram desenvolvidos pensando na versão 3.6 da linguagem Python.

Sumário

Algoritmo genético 1

1 Introdução 2

1.1 Máquina de Turing 2
1.2 Preparando o ambiente 6
1.3 Algoritmo genético: conceitos gerais 8

2 Produção e avaliação de indivíduos 13

2.1 Criando uma população 13
2.2 Função de avaliação 19

3 Seleção 30

3.1 Seleção via roleta 34
3.2 Seleção por classificação 40
3.3 Seleção por torneio 43

4 Cruzamento 49

4.1 Cruzamento de um ponto 54
4.2 Cruzamento em k-pontos 57
4.3 Embaralhamento 64

5 Mutação **70**

5.1 Mutação flip 75
5.2 Mutação de dupla troca 77
5.3 Mutação de sequência reversa 79

6 Evolução **83**

6.1 A sobrevivência do mais adaptado: integrando os operadores genéticos 84
6.2 Encontrando o máximo 89

7 A evolução através da tragédia **98**

7.1 Operador Epidêmico 99

8 Disponibilização do pacote **105**

8.1 Organização do pacote 105
8.2 Subindo o pacote para o PyPi 107

Estudo de caso **109**

9 Explorando o uso de Algoritmos Genéticos **110**

10 Gerando labirintos **112**

10.1 Grafos 112
10.2 Gerador de labirinto 115

11 Da imagem ao grafo **125**

11.1 Do pixel ao vértice 126

12 Encontrando caminhos **134**

12.1 Avaliação de indivíduo 134

12.2 Algoritmo genético 141

13 Conclusões 149

13.1 Referências bibliográficas 149

Algoritmo genético

Capítulo 1
INTRODUÇÃO

A computação evolucionária tem como base filosófica a teoria da sobrevivência do melhor adaptado de Darwin (1859). Uma das primeiras aplicações da teoria biológica da evolução para algoritmos computacionais está associada ao trabalho de Holland (1975) e seus colaboradores. Para entender um pouco melhor a computação evolucionária, primeiramente temos que ter em mente o significado de *algoritmo* e *computação*. Antes de mais nada é fundamental levantar alguns conceitos de teoria da computação.

1.1 MÁQUINA DE TURING

A Máquina de Turing é um computador abstrato formado por uma fita, representando a memória, que tem um começo de um lado, e que é infinita do outro, um cabeçote para ler e escrever na fita. Essa máquina é usada para representar o processo de computação. A partir dela é possível determinar os limites do que pode ser resolvido através do uso de computadores. Assim, a definição formal de algoritmo é: conjuntos de regras que podem ser executados por alguma máquina de Turing. Se para um dado problema não existir uma máquina de Turing que o resolva, então não existe um algoritmo que o solucione.

Turing mostrou que sua máquina abstrata é equivalente ao Cálculo-Lambda de Alonzo Church. Com isso, a definição formal de algoritmo, em termos da máquina de Turing, ficou conhecida como Tese de Church-Turing. O termo tese aqui é usado para indicar que, até o momento, não sabemos se existe um modelo computacional mais potente que a máquina de Turing. Mesmo um computador quântico seria equivalente a uma máquina de Turing estocástica, a qual pode ser reduzida a uma máquina de Turing comum. Existem trabalhos tentando encontrar problemas que só poderiam ser resolvidos via computação quântica, mas isso ainda é um campo em aberto.

A partir do trabalho de Alan Turing (1936) passou a ficar claro quais os limites da computação, sendo possível provar, utilizando técnicas formais, que certos problemas não podem ser resolvidos a partir de algoritmos. Com esse trabalho nascia a Ciência da Computação como um ramo completamente novo.

Um caso que não tem solução algorítmica é o chamado *problema da parada*, o qual consiste em escrever um programa capaz de verificar se um outro programa vai executar para sempre, como em um laço infinito, ou não.

Máquina de Turing determinística

De forma geral, uma máquina de Turing determinística é aquela que reconhece uma linguagem (resolver um problema) de forma bem determinada, como em programação estruturada, seguindo um passo de cada vez.

Máquina de Turing não determinística

A máquina de Turing não determinística é uma máquina capaz de explorar várias soluções possíveis de forma simultânea, realizando a busca de todas as soluções determinísticas de uma só vez. Esse efeito é similar ao que se espera de um computador quântico.

Tempo polinomial

Um polinômio é uma expressão matemática do tipo $a_0x^0+a_1x^1+...+a_kx^k$, sendo k o maior termo do polinômio. Se para uma entrada de tamanho n o número de passos para chegar a uma resposta puder ser descrito por um polinômio, então dizermos que o tempo de processamento do algoritmo é polinomial.

A questão de P versus NP

Em Ciência da Computação, um problema é chamado do tipo P (Polinomial) se existe uma máquina de Turing determinística capaz de resolvê-lo em tempo polinomial. Por exemplo, ordenar lista de números é um problema relativamente simples, no qual,

usando poucos laços de repetição, é possível criar um programa para resolvê-lo. Por outro lado, problemas NP são aqueles que podem ser resolvidos em tempo polinomial somente usando uma máquina de Turing não determinísticas (o N do NP vem de "não determinísticas"). São problemas difíceis de resolver, tomando muito tempo para ser solucionado (de horas a séculos) em um computador determinístico. Problemas NP-completos são aqueles que podem ser solucionados de forma equivalente, ou seja, o mesmo perfil de algoritmo é capaz de resolver um conjunto de problemas do tipo NP. Esse processo de converter um problema em outro é chamado de **redutibilidade**.

Um grande desafio da atualidade é saber se existe uma forma mais fácil de resolver problemas do tipo NP, o que indicaria que esses problemas são do tipo P. Muitos algoritmos de criptografia são do tipo NP, logo, se P é igual a NP, então os programas de criptografia poderiam ser facilmente quebrados, resolvidos em tempo polinomial por computadores ordinários.

O outro caso da quebra de criptografia está ligado à computação quântica, pois esses computadores seriam equivalentes à máquina de Turing não determinística, sendo capazes de resolver de forma fácil os problemas do tipo NP.

Quando um problema possui solução, mas sendo difícil de ser encontrada, podemos ter algoritmos genéticos como ferramenta para resolver o problema, pois eles pertencem à categoria chamada de busca estocástica. Tais algoritmos mapeiam de forma semialeatórias o espaço de soluções possíveis, buscando bons candidatos para resolver um dado problema.

Por exemplo, imagine que uma empresa está desenvolvendo

um novo satélite, cujo espaço é limitado. Nesse caso é preciso determinar qual a melhor forma de dispor o máximo possível de equipamentos e sensores, ao mesmo tempo que é preciso manter o centro de equilíbrio de gravidade do satélite em seu centro geométrico. Uma solução seria a de testar todos os arranjos possíveis de distribuição, escolhendo as que geram a saída esperada. Usando computação evolucionária, podemos pensar em distribuições de equipamentos como se fossem seres de alguma espécie misteriosa. Os mais adaptados serão aqueles que fornecem o centro de gravidade do satélite o mais próximo possível do seu centro geométrico. Podemos então combinar distribuições dos melhores indivíduos, como por exemplo, usando a localização de cinco equipamentos de um e mais cinco de outros, escolhidos aleatoriamente, e verificar se a distribuição melhorou. Repetindo esse processo por algumas gerações, vamos encontrar a melhor distribuição possível de equipamentos.

1.2 PREPARANDO O AMBIENTE

Neste livro, vamos explorar os conceitos de Algoritmos Genéticos (AG), os quais fazem parte da computação evolucionária. Será apresentado o processo de criação de um framework que possa ser usado de forma generalizada. Para isso, serão explorados de forma profunda conceitos de Programação Orientada a Objetos e a utilização da biblioteca Numpy, a qual permite estender a linguagem Python para incluir processamento numérico e vetorial, além de utilização da biblioteca gráfica, matplotlib para geração de gráficos e animações.

A estrutura básica de pastas que vamos usar no projeto será a seguinte:

```
|-pygenec_pypi\
              |-env\
              |-src\
                    |-pygenec\
                              |-selecao\
                              |-cruzamento\
                              |-mutacao\
```

A pasta `env` vai conter o ambiente virtual Python. A vantagem aqui está em criar uma área de trabalho independente, evitando conflitos gerados por uso de versões diferentes de bibliotecas. Para mais informações, visitar https://virtualenv.pypa.io/. Com o Python já instalado na máquina, vamos adicionar o `virtualenv` , usando o seguinte comando, no terminal:

```
python -m pip install virtualenv
```

A criação do ambiente virtual é realizada pelo comando:

```
virtualenv -p python3.6 env
```

O comando anterior vai criar a pasta `env` contendo a versão 3.6 do Python, a qual já deve estar previamente instalada na máquina.

Para ativar o ambiente de trabalho, usamos o comando, no Linux:

```
source env/bin/activate
```

Após ativar o ambiente virtual, vamos instalar as bibliotecas básicas que utilizaremos ao longo desse projeto, que são o numpy e o matplotlib.

Numpy é uma biblioteca para computação científica em Python, que permite a integração com Fortran e C. Extremamente

útil para trabalhar com vetores de n-dimensões, contendo uma séria de métodos para álgebra linear, transformadas de Fourier e geração de números aleatórios. Para mais informações, visite: http://www.numpy.org/.

Para instalar a biblioteca numpy, usamos o comando:

```
pip install numpy
```

Matplotlib é uma biblioteca para desenvolvimento de gráficos (*plotting*) em 2d e 3d. Para mais informações, visite: https://matplotlib.org.

Sua instalação é possível via comando:

```
pip install matplotlib
```

O framework de AG desenvolvido aqui vai seguir o conceito de cadeias de bits, os quais representarão possíveis soluções de um dado problema, tal qual foi apresentado por Holland (1975). Os bits serão chamados de genes e seus conjuntos, o quais podem ser traduzidos para um valor numérico, como na posição do centro de um equipamento dentro de um satélite, por exemplo, serão chamados de cromossomos.

1.3 ALGORITMO GENÉTICO: CONCEITOS GERAIS

Como vimos, o AG tem como fundamento a continuidade das melhores soluções de um dado problema. Tal como na natureza, os indivíduos que mais adaptados ao seu ambiente terão maior probabilidade de procriar. Esse perfil de seleção é chamado de elitista.

Em linhas gerais, as etapas do AG são:

1. criação de uma população inicial;
2. determinação dos indivíduos mais adaptados;
3. cruzamento e procriação;
4. mutação.

Pode-se ter variações de como são realizadas cada uma das etapas. Assim, teremos diferentes tipos de operadores genéticos. Ao longo do livro iremos explorar ao menos três tipos de operadores de cruzamento, seleção e mutação.

População

O primeiro passo consiste em determinar o número de cromossomos e genes que são capazes de representar uma possível solução de um problema. Como exemplo, considere que queremos achar o máximo de uma função f(x,y). Nesse caso, temos dois parâmetros x e y, assim teremos dois cromossomos a serem considerados. Agora, suponha que queremos determinar o máximo da função para valores de x que estejam entre 1 e 5 e para valores de y que estão entre 10 e 20. O número de genes para cada cromossomo vai determinar o menor intervalo de busca possível.

Por exemplo, vamos assumir que vamos utilizar 8 bits para cada cromossomo, para o caso de x, o menor intervalo de busca será dado por (5-1) / 256 = 0,015625. Note que 8 bits conseguem representar números que vão entre 0 a 255, o que dá 256 representações numéricas. Quanto mais genes tiver um cromossomo, mais refinada será a busca, dentro de um intervalo.

Seleção

Como cada indivíduo pode ser traduzido em uma solução do problema é preciso escolher as melhores. Esse procedimento é chamado de seleção. Podemos definir probabilidades diferentes para que um indivíduo seja escolhido, de acordo com o quão bom ele representa uma solução. O valor numérico da qualidade do indivíduo é chamado de *fitness*. Quanto maior o *fitness* do indivíduo, maior será a sua possibilidade de procriação.

Cruzamento

Após definir o critério de seleção, o próximo passo será o de combinar as cadeias genéticas. Dois indivíduos são selecionados e sua carga genética é combinada, gerando dois novos filhos. Esse procedimento deverá ser repetido até se ter um número total de novos indivíduos igual ao tamanho da população original menos um. No modelo elitista, os genes do melhor indivíduo da população velha são mantidos inalterados na população nova. Com isso, a nova população sempre terá o tamanho total da população antiga.

Mutação

A mutação tem um papel importante no processo de evolução. Através de pequenas mudanças, novas características podem ser introduzidas. Em alguns casos, essa modificação poderá melhorar a adaptabilidade de uma espécie ao meio. Mas o fato é que a maior parte da mutação não será benéfica. Porém, todo novo indivíduo está sujeito a alguma transformação de seu código genético. Com isso, após gerar novos indivíduos, aplicamos o operador de

mutação e escolhemos indivíduos aleatoriamente para sofrer alguma modificação.

Evolução

A evolução consiste em aplicar os passos de seleção, cruzamento e mutação a cada nova população durante um número de iterações predefinidas pelo desenvolvedor. Ao final dos ciclos, espera-se ter encontrado indivíduos que representam a melhor solução para um dado problema. Na figura a seguir é apresentado o esquema geral do AG.

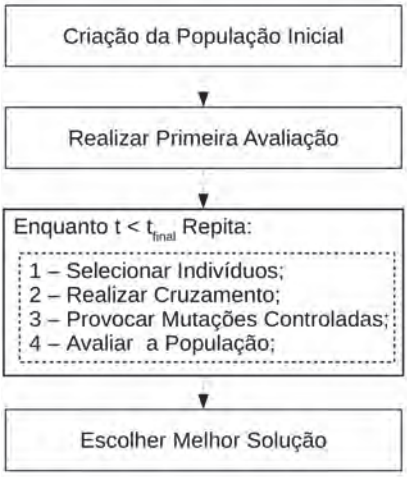

Criação da População Inicial
Realizar Primeira Avaliação
Enquanto t < t_final Repita:
1 – Selecionar Indivíduos;
2 – Realizar Cruzamento;
3 – Provocar Mutações Controladas;
4 – Avaliar a População;
Escolher Melhor Solução

Figura 1.1: Representação Esquemática do funcionamento do Algoritmo Genético.

Conclusões

Neste capítulo focou em trazer uma breve introdução aos conceitos de computação evolucionária e Algoritmos Genéticos (AGs). Os quais são do tipo estocástico e empregados

principalmente em problemas difíceis de resolver usando computadores. Além desses casos, sua utilização em problemas com caráter mais dinâmico, como em sistemas de previsão de séries temporais pode ser bastante útil. O livro é dividido em duas partes: aqui na primeira, será desenvolvido um framework para a criação de AGs. A segunda parte envolve a aplicação de AG, em particular, utilizaremos a busca por caminhos em labirintos para explorar conceitos de grafos, redutibilidade de problemas e utilização mais avançadas da biblioteca gráfica matplotlib.

Capítulo 2

PRODUÇÃO E AVALIAÇÃO DE INDIVÍDUOS

O cerne do algoritmo genético está na evolução de uma população de indivíduos. Logo, o primeiro passo será o de criar a nossa população. Cada indivíduo deve ser avaliado, de acordo com o problema que se quer resolver. Ou seja, vamos comparar as características de um ente a uma dada condição, o que gerará uma classificação geral. Com isso, poderemos saber quem está mais bem qualificado para gerar descendentes, ou mais bem adaptado às condições dadas pelo problema que se quer resolver.

Aqui vamos manter o número total da população constante ao longo de todas as gerações.

Cada indivíduo será composto por uma cadeia genética formada por zeros (0) e uns (1), os quais serão representados por vetores, usando a biblioteca Numpy.

2.1 CRIANDO UMA POPULAÇÃO

Vamos começar criando a estrutura do projeto. Inicialmente, a pasta raiz terá a seguinte hierarquia:

```
pygenec\
```

```
|
|-pygenec\
          |-__init__.py
          |-populacao.py
|-main.py
```

Nosso projeto se chamará `pygenec`, e conterá um pacote com o mesmo nome. O arquivo `__init__.py` é usado para identificar o pacote, o qual contém o arquivo `populacao.py`. No arquivo `main.py` vamos colocar a função de avaliação, apresentando uma aplicação do algoritmo genético.

Usando um editor de texto de sua preferência, abra o arquivo `populacao.py` e adicione a seguinte classe:

```
#!/usr/bin/env python3.6
# -*- Coding: UTF-8 -*-
"""
Gerador aleatório de população.

Programa sob licença GNU V.3.
Desenvolvido por: E. S. Pereira.
Versão 0.0.1.
"""

from numpy.random import randint
from numpy import argsort, unique


class Populacao:
    """
    Cria e avalia uma população.
    Recebe como entrada:
        avaliacao - Função que recebe um indivíduo como entrada e
 retorna
                    um valor numérico.
        cromossos_totais - Número inteiro representando o tamanho
 da cadeia
                           genética do indivíduo.
        tamanho_populacao - Número inteiro representando o número
 total de
                            indivíduos na população.
```

```
    """

    def __init__(self, avaliacao, genes_totais, tamanho_populacao
):
        self.avaliacao = avaliacao
        self.genes_totais = genes_totais
        self.tamanho_populacao = tamanho_populacao

    def gerar_populacao(self):
        """Gerador aleatório de população."""
        self.populacao = randint(0, 2,
                                 size=(self.tamanho_populacao,
                                       self.genes_totais),
                                        dtype='b')

     def avaliar(self):
         """Avalia e ordena a população."""
         u, indices = unique(self.populacao, return_inverse=True,
 axis=0)
         valores = self.avaliacao(u)
         valores = valores[indices]
         ind = argsort(valores)
         self.populacao[:] = self.populacao[ind]
         return valores[ind]
```

Na primeira linha apenas indicamos para sistemas tipo Unix que o programa deve ser executado pela versão 3.6 do Python. Já na segunda, identificamos que estamos usando a codificação UTF-8 para o código escrito no arquivo. Em seguida é adicionado o comentário de documentação do módulo.

Ao longo de todo o livro, vamos usar bastante o **numpy**. Suas funções são fundamentais para trabalhar com vetores e matrizes, além das diversas funções de matemática de alto nível. Nesse módulo, importamos a função `randint`, para gerar números inteiros aleatórios dentro de um intervalo. Também vamos usar a função `argsort`, a qual retorna o índice de um vetor ordenado de forma crescente. Por exemplo:

```
from numpy import array, argsort
a = array([3, 4, 1, 7, 5, 2])
print(argsort(a))
# array([2, 5, 0, 1, 4, 3])
```

O `argsort(a)` vai retornar um vetor cujo primeiro número é 2, ou seja, indicando que no vetor `a` o menor valor numérico está na posição 2. Aqui vale ressaltar que a posição inicial do vetor é 0.

Na sequência, criamos a classe `Populacao`, cujo construtor é dado pelo método `__init__`. A classe recebe a função de avaliação (`avaliacao`), o número total de genes de cada indivíduo (`genes_totais`) e o tamanho total da população (`tamanho_populacao`).

Para começar o projeto, vamos utilizar a função `randint`, que é capaz de gerar um vetor de números aleatórios, recebendo parâmetros da seguinte forma: `randint(inferior, superior, tamanho, dtype)`, sendo que inferior representa o menor valor de inteiro a ser incluído no vetor; superior, o maior valor a ser considerado, mas que não será incluído no vetor; tamanho indica o tamanho do vetor, podendo ter mais de uma dimensão, similar a uma matriz; já o parâmetro `dtype` indica o tipo de dado, como `'int'` para inteiro e `'b'` para binário. O uso dessa função é apresentado dentro do método `gerar_populacao`. Assim, o atributo `populacao` será inicializado cada vez que o método for chamado.

Um exemplo de uma população gerada aleatoriamente é apresentado a seguir. Nesse caso, cada linha da matriz representa um indivíduo, sendo que cada coluna é um gene. Aqui temos 5 indivíduos, e cada um possui 8 genes. Logo, a população será representada por uma matriz de 5x8.

```
[[1 0 1 0 1 0 0 1]
 [1 0 0 0 0 0 0 1]
 [0 1 0 1 0 0 0 1]
 [0 0 1 1 0 0 1 1]
 [1 1 1 1 1 0 0 0]]
```

O método `avaliar` usa a função de avaliação para atribuir valores numéricos aos indivíduos da população (`valores = self.avaliacao(self.populacao)`). A variável `valores` representa um vetor com as avaliações da população. O nosso próximo passo será o de ordenar a população em ordem crescente, de acordo com os resultados obtidos pela função `avaliacao` . Para isso usamos o `argsort` :

```
ind = argsort(valores)
```

Note que a variável `ind` contém os índices do vetor `valores` ordenados respeitando o resultado obtido via função de avaliação.

Usando a variável `ind` fazemos a ordenação dos indivíduos da população:

```
self.populacao[:] = self.populacao[ind] .
```

Com o comando `self.populacao[:]` , indicamos que queremos utilizar o mesmo espaço de memória, utilizado na criação da população inicial. Nesse caso, o que vai ocorrer é que cada elemento de linha e coluna representando a população anterior será substituído pelos valores da população ordenada, de acordo com a variável `ind` .

Observe que o numpy aceita uma lista de índices para operar um vetor sem a necessidade de se criar um laço de repetição. Dessa forma, conseguimos criar operações capazes de acessar todos os elementos de um vetor usando apenas um único comando. Esse foi

o caso de se utilizar `self.populacao[ind]`, o qual retornará o vetor `self.populacao` respeitando a ordem descrita pela lista de índices contidas em `ind`.

A vantagem de se fazer isso é que estamos economizando memória. Por exemplo, poderíamos ter utilizado o seguinte comando: `self.populacao = self.populacao[ind]`. Nesse caso, estaríamos indicando que a variável `self.populacao` deveria apontar para o novo vetor, representado pela população ordenada. A diferença no comando é muito pequena, apenas retiramos o `[:]` após `self.populacao`.

Essa diferença ocorre devido ao fato de que, em Python, as variáveis são passadas por referência. Isso quer dizer que uma variável aponta sempre para um endereço de memória e não para um valor em si. Considere o seguinte exemplo:

```
>>> a = [1, 2, 3]
>>> b = a
>> b[0] = 2
>> print(a)
[2, 2, 3]
>> b = a[:]
>> b[0] = 3
>> print(a)
[2, 2, 3]
>> print(b)
[3, 2, 3]
>>> b = [6, 7, 8]
```

Observe que, quando fazemos `b = a`, `b` aponta para o mesmo endereço de memória de que `a`. Com isso, modificar `b` acaba por modificar `a`. Fazendo `b = a[:]` indicamos que `b` será uma cópia da lista em `a`, então modificar `b` não interfere mais em `a`. Na última linha, ao fazer `b = [6, 7, 8]`, a variável `b` passa a apontar para uma nova lista. A lista antiga `[3, 2, 3]`,

para a qual `b` apontava previamente, ficará na memória, não sendo eliminada imediatamente. Em um determinado momento futuro, o *garbage collector* (coletor de lixo) do Python apagará essa lista órfã. Porém, não temos controle sobre quando isso vai acontecer. Isso pode acabar gerando muito lixo na memória do computador. Logo, ao se reutilizar a mesma estrutura de dados conseguimos reduzir o uso de memória pelo nosso programa.

2.2 FUNÇÃO DE AVALIAÇÃO

Segundo Kiusalaas (2005), optimização significa encontrar o mínimo ou o máximo de uma função, sendo que tal função é chamada de função mérito ou função objetivo. A função de avaliação será determinada pelo problema que se quer resolver. Vamos considerar inicialmente que desejamos encontrar o máximo da função, representada pela figura a seguir:

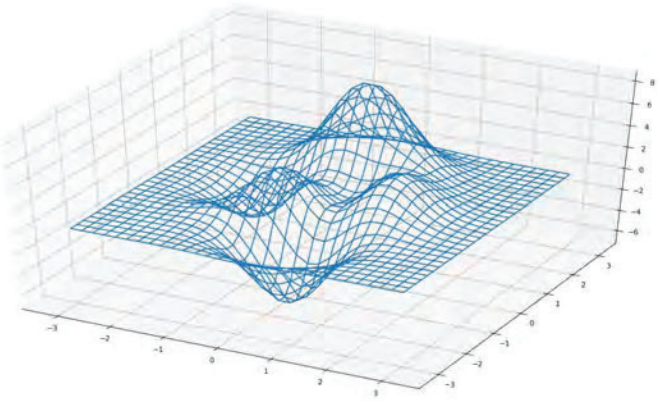


Figura 2.1: Função objetivo

A figura anterior foi gerada usando a seguinte função:

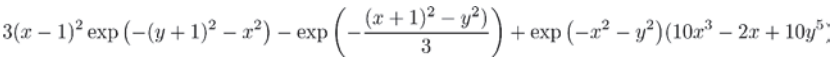


Figura 2.2: Função objetivo

Um primeiro objetivo será o de utilizar o algoritmo genético para determinar o máximo da função anterior. Assim, essa será a nossa função objetivo.

A função de avaliação será construída tendo como base a função objetivo. Como queremos encontrar o máximo da função, o indivíduo melhor adaptado será aquele que, ao ser contraposto contra a função objetivo, retorna o maior valor possível.

Vamos editar o arquivo `main.py`, para representar o nosso problema de otimização. Inicialmente, importamos as funções e classes necessárias, como a função de expoente natural e a classe `Populacao`, criada anteriormente.

```
from numpy import exp, array
from pygenec.populacao import Populacao
```

Em seguida, vamos escrever a função objetivo:

```
def fun(x, y):
    tmp = 3 * exp(-(y + 1) ** 2 - x **2)*(x - 1)**2 \
          - (exp(-(x+ 1) ** 2 - y **2) / 3 )\
          + exp(-x **2 - y ** 2) * (10 * x **3 - 2 * x + 10 * y *
* 5)
    return tmp
```

Observe que a função objetivo, `fun`, recebe dois parâmetros, `x`, `y`. Dessa forma, vamos dividir os genes dos indivíduos em dois cromossomos. O número de estruturas genéticas, ou cromossomos, que vamos usar está diretamente ligado à quantidade de parâmetros da função objetivo.

Os cromossomos dos indivíduos são formados por números binários. Logo, teremos que separar os genes em cromossomos. Em seguida, cada cromossomo deverá ser convertido para um número inteiro, para ser usando na função objetivo. A conversão de binário para inteiro é feita pelo seguinte somatório:

$$I=2^0b[0]+2^1b[1]+...+2^ib[i]$$

Sendo `i=n-1`, em que `n` é o tamanho do vetor `b`, contendo `0` ou `1` em cada posição.

A função de conversão será:

```
def bin(x):
    cnt = array([2 ** i for i in range(x.shape[1])])
    return array([(cnt * x[i,:]).sum() for i in range(x.shape[0])
])
```

Note que `cnt` representa o vetor com as potências de base 2. A operação `cnt * x[i,:]` gera um novo vetor em que cada elemento `i` representa a multiplicação dos elementos pareados de `cnt` e `x`. Ou seja, o primeiro elemento do vetor resultante é dado pela multiplicação do primeiro elemento de `cnt` com primeiro elemento de `x`, assim sucessivamente.

Quando se trata de processo de otimização, precisamos definir um espaço de busca. Isso significa que temos que definir qual a região, dentro da função, em que estamos interessados. Existe também a questão de custo computacional: como uma função pode descrever um espaço infinito, sem a restrição, a busca poderá nunca ter fim. Por exemplo, imagine que se quer reduzir os custos de produção de um veículo. Existem muitos parâmetros a se considerar, como custo de mão de obra, preço de metal e tina, custo ambiental e muitos outros itens. Sabemos também que

insumos muito baratos em geral costumam ter uma qualidade pior. Ao mesmo tempo, usar apenas insumos caros reduziria muito o lucro final. Em um processo de otimização, acabamos aplicando restrições aos valores dos insumos e fazendo a busca dentro de um intervalo. Nesse exemplo, a função objetivo será aquela que retorna o maior lucro possível, com o menor gasto de material de produção, sendo que o produto final deverá ser adequado às normas de qualidade.

Tal como no exemplo anterior, vamos restringir o intervalo de busca da função, dessa forma, o número inteiro gerado anteriormente precisa ser mapeado para o intervalo de interesse, que ficará entre -3 e 3. Fazemos isso com a seguinte função:

```
def xy(populacao):
    colunas = populacao.shape[1]
    meio = colunas // 2
    maiorbin = 2.0 ** meio - 1.0
    nmin = -3
    nmax = 3
    const = (nmax - nmin) / maiorbin
    x = nmin + const * bin(populacao[:,:meio])
    y = nmin + const * bin(populacao[:,meio:])
    return x, y
```

A função `xy` recebe uma população como parâmetro. Internamente, a primeira coisa que é feita é a obtenção do número total de colunas desse vetor (tamanho do indivíduo). Como foi dito anteriormente, metade da codificação genética será usada para representar a variável `x`, enquanto a outra metade representará a variável `y`. Observe que, a partir do total de colunas da matriz `populacao`, `meio = colunas // 2` realiza a divisão dos genes em dois conjuntos de cromossomos.

Note que `maiorbin = 2.0 ** meio - 1.0` representa o

maior número inteiro capaz de ser escrito usando um número binários tendo um total de dígitos contidos na variável `meio`. Por exemplo, considere um indivíduo com dezesseis genes. Nesse caso, `meio=8`. O maior número inteiro que conseguimos representar com 8 bits é 2^8-1 que é 255.

As variáveis `nmin` e `nmax` representam o intervalo de busca. Isso quer dizer que vamos procurar o máximo da nossa função apenas dentro do intervalo de -3 até 3.

Precisamos garantir que os valores gerados a partir da população estejam dentro desse intervalo, então precisamos normalizar os valores obtidos dos cromossomos para o intervalo de interesse. Para fazer isso, criamos uma constante de normalização, que nada mais é do que a divisão do intervalo de interesse pelo maior número inteiro capaz de ser representado pelo binário contendo o total de bits presentes na variável `meio`. A criação da constante de normalização é realizada a partir do comando `const = (nmax - nmin) / maiorbin`.

O mapeamento final do código genético do indivíduo em intervalo de busca da função é dado pelas declarações `x = nmin + const * bin(populacao[:,:meio])` e `y = nmin + const * bin(populacao[:,meio:])`.

Aqui vale ressaltar que o tamanho da cadeia genética vai representar a resolução da busca, logo, quanto maior a cadeia genética, mais números serão representados dentro do intervalo. Por exemplo, uma cadeia de 4 bits consegue representar 16 números no intervalo de 0 a 1, já com uma cadeia de 8 bits conseguimos representar 256 números nesse mesmo intervalo.

Para o caso de problema de buscas de mínimo e máximo, uma resolução menor pode ser interessante para evitar mínimos ou máximos locais, porém poderá ocultar um verdadeiro mínimo global.

Finalmente, na função `avaliacao` usamos os valores `x` e `y` para calcular a função objetivo. Como `x` e `y` são vetores, a varável `tmp` também será um vetor, de números reais, do tamanho da população, o que é dado pelo seguinte código:

```
def avaliacao(populacao):
    x, y = xy(populacao)
    tmp = func(x, y)
    return tmp
```

Para visualizar o resultado, vamos usar a biblioteca de geração de gráficos chamada **matplotlib**. No topo do arquivo `main.py` vamos importar os módulos necessários:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
from numpy import mgrid
```

O módulo `pyplot` é o principal, contendo as funções para gerar os gráficos. A função `axes3d` é importada para indicar ao matplotlib que queremos gerar gráficos 3d. A função `mgrid` será utilizada na geração de grades retangulares, as quais serão usadas para construir as bases do gráfico 3d.

Para criar uma população com oito indivíduos, em que cada um tem quatro genes:

```
genes_totais = 8
tamanho_populacao = 5

populacao = Populacao(avaliacao, genes_totais, tamanho_populacao)
populacao.gerar_populacao()
```

Em seguida, será executado o método `avaliar`. Também vamos produzir os valores `x` e `y`, a serem usados na função de avaliação, a partir dos cromossomos da população.

```
populacao.avaliar()
x, y = xy(populacao.populacao)
```

Para gerar o gráfico da função objetivo e da posição dos indivíduos usaremos o seguinte código:

```
fig = plt.figure(figsize=(100, 100))
ax = fig.add_subplot(111, projection="3d")
X, Y = mgrid[-3:3:30j, -3:3:30j]
Z = func(X,Y)
ax.plot_wireframe(X, Y, Z)
ax.scatter(x, y, func(x, y), s=50, c='red', marker='D')
plt.show()
```

O comando `fig = plt.figure(figsize=(100, 100))` gera a estrutura básica do gráfico, que terá o tamanho de 100x100 polegadas. Na variável `ax` adicionamos um subgráfico com projeção em 3d. Para esse gráfico, criamos uma grade `X`, `Y` nos intervalos de -3 até 3 com 30 pontos entre esses valores. A escolha do número de pontos é arbitrária: quanto mais pontos usarmos, maior será a resolução do gráfico, o que leva a um maior custo computacional para a geração da imagem final.

A variável `Z` representa os valores da função objetivo dentro da grade `X`, `Y`. O comando `ax.plot_wireframe(X, Y, Z)` cria um gráfico 3d do tipo arame, que gera uma superfície de linhas entrelaçadas. Ele será interessante para o nosso caso, pois isso facilitará a visualização dos pontos que representam os valores obtidos a partir dos indivíduos da população.

Com o comando `ax.scatter(x, y, func(x, y), s=50, c='red', marker='D')` adicionamos um gráfico de pontos

espalhados, representando os valores calculados de `x`, `y` e da função objetivo para cada indivíduo de forma distinta. Para exibir o gráfico, usamos o comando `plt.show()`.

Na próxima figura é apresentado o resultado da avaliação. Os pontos vermelhos na grade representam os indivíduos da população. A ideia do algoritmo genético é que, com o tempo, a população comece a convergir para o ponto mais alto nessa superfície. Imagine que o topo da superfície represente maior abundância de alimento. Os indivíduos que estão nas regiões mais elevadas terão maior chance de sobreviver. Consequentemente, esses indivíduos terão maior probabilidade de procriar. Como o sucesso de um indivíduo será dado pelo quão alto ele está, o processo de evolução fará com que indivíduos no topo tenham maior chance de procriar.

Como a população é gerada aleatoriamente, cada vez que o programa for executado teremos resultados diferentes, para a posição dos indivíduos com relação à função de avaliação.

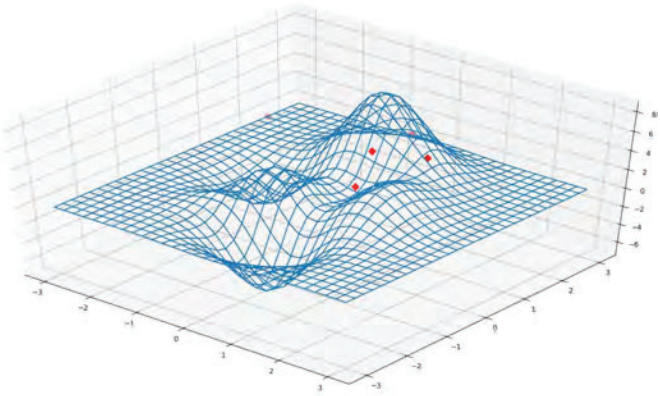


Figura 2.3: Função objetivo e soluções de busca por máximo.

Se o interesse for encontrar o mínimo no lugar do máximo, basta mudar a função `avaliacao` para calcular o seu negativo:

```
def avaliacao(populacao):
    #Calcula o mínimo da função objetivo
    x, y = xy(populacao)
    tmp = - func(x, y)
    return tmp
```

O código completo da função `main.py` é apresentado a seguir:

```
#!/usr/bin/env python3.6
# -*- Coding: UTF-8 -*-
"""
Obetenção de máximo de função.

Programa sob licença GNU V.3.
Desenvolvido por: E. S. Pereira.
Versão 0.0.1.
"""
from numpy import exp, array, mgrid
from pygenec.populacao import Populacao

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d


def func(x, y):
    tmp = 3 * exp(-(y + 1) ** 2 - x **2)*(x - 1)**2 \
          - (exp(-(x+ 1) ** 2 - y **2) / 3 )\
          + exp(-x **2 - y ** 2) * (10 * x **3 - 2 * x + 10 * y *
* 5)
    return tmp


def bin(x):
    cnt = array([2 ** i for i in range(x.shape[1])])
    return array([(cnt * x[i,:]).sum() for i in range(x.shape[0])
])

def xy(populacao):
    colunas = populacao.shape[1]
```

```
    meio = colunas // 2
    const = 2.0 ** meio - 1.0
    nmin = -3
    nmax = 3
    const = (nmax - nmin) / const
    x = nmin + const * bin(populacao[:,:meio])
    y = nmin + const * bin(populacao[:,meio:])
    return x, y

def avaliacao(populacao):
    x, y = xy(populacao)
    tmp = func(x, y)
    return tmp

cromossos_totais = 8
tamanho_populacao = 5

populacao = Populacao(avaliacao, cromossos_totais, tamanho_popula
cao)
populacao.gerar_populacao()
populacao.avaliar()

x, y = xy(populacao.populacao)

fig = plt.figure(figsize=(100, 100))
ax = fig.add_subplot(111, projection="3d")
X, Y = mgrid[-3:3:30j, -3:3:30j]
Z = func(X,Y)

ax.plot_wireframe(X, Y, Z)
ax.scatter(x, y, func(x, y), s=50, c='red', marker='D')
plt.show()
```

Conclusão

Neste capítulo foi apresentada a criação da classe `Populacao`, a qual é utilizada para gerar uma população de indivíduos, cuja codificação genética é representada por `0 e 1`.

Também se mostrou como converter uma função objetivo, para a qual se deseja encontrar o valor máximo, na função de

avaliação, que é utilizada para classificar e ordenar a população.

O próximo capítulo tratará do processo de seleção, o qual será usado para fazer o cruzamento entre as soluções.

Capítulo 3

SELEÇÃO

Nosso algoritmo é inspirado no processo de evolução. Na natureza, os indivíduos mais adaptados às condições ambientais têm maior probabilidade de se reproduzir, passando seu código genético adiante. No processo de criação de indivíduos, o passo inicial era criar uma população com genes aleatórios. A partir da seleção, começamos a separar os candidatos como maior potencial para se alcançar uma solução ótima para o problema que se quer resolver.

Após a geração da população, o próximo passo é o de selecionar indivíduos para a reprodução. Tal seleção é baseada na classificação realizada através da função de avaliação, a qual foi apresentada no capítulo anterior.

A seleção deve ser executada de tal forma que não elimine por completo a possibilidade de escolha de soluções ruins. Como estamos trabalhando com o fator *acaso*, às vezes, cruzamento de soluções ruins poderá gerar soluções boas. Além disso, em geral, trabalhamos com populações não muito grandes, por questão de custo computacional. Nesse caso, a seleção deve permitir também a variabilidade genética da população, pois isso ajuda a ampliar o espaço de busca. Mas se a seleção não for adequadamente balanceada, o processo de busca poderá ficar muito lento,

demandando muito tempo para se alcançar o resultado esperado.

Alguns dos métodos de seleção mais comuns são (ver Saini, 2017): Seleção via roleta; Seleção por classificação, Seleção por torneio.

Como não existe apenas um método de seleção, vamos aproveitar o poder da programação orientada a objetos e criar uma classe, contendo métodos abstratos (classe mãe), que será usada para definir o padrão das chamadas de métodos, a nossa API no processo de construção do algoritmo genético.

> API (*Application Programming Interface*) - Interface de Programação de Aplicação: é um padrão de projeto usado quando se deseja que outros componentes consigam usar partes de um outro software, ou de um objeto, sem ter que se preocupar com o funcionamento interno de um método ou função. Os nomes dados aos métodos precisam ter clareza suficiente para que um usuário possa inferir, a partir do nome, qual o resultado esperado para uma determinada chamada de método ou função. O termo API não se resume somente à criação de aplicações web.

Vamos construir uma classe mãe para a implementação dos operadores de seleção. Na pasta `pygenec` vamos criar a pasta `selecao`, contendo um arquivo em branco chamado `__init__.py` e o arquivo `selecao.py`.

```
pygenec\
 |
```

```
|-pygenec\
          |-__init__.py
          |-populacao.py
          |-selecao\
                    |-__init__.py
                    |-selecao.py
|-main.py
```

A classe terá dois métodos públicos, `selecionar`, que retornará o índice do indivíduo selecionado, e `selecao`, que recebe como entrada o tamanho total da seleção e que retorna uma subpopulação. Do `numpy` vamos importar a função `array`. Já o construtor receberá um objeto do tipo `Populacao`:

```
from numpy import array


class Selecao:
    """
    Seleciona indivíduos para cruzamento.
    Recebe como entrada:
        populacao - Objeto criado a partir da classe Populacao.
    """
    def __init__(self, populacao):
        self.populacao = populacao
```

O método `selecionar` vai levantar um erro de implementação, pois ele será modificado ao se construir a classe de cada operador específico:

```
    def selecionar(self, fitness):
      """
      Retorna a lista de índice do vetor população
      dos indivíduos selecionados.
      """
      raise NotImplementedError("A ser implementado")
```

Com isso, estamos fazendo com que o método `selecionar` seja abstrato, pois só poderá ser usado quando a classe filha efetivamente o implementar. Ao mesmo tempo, o nome

`selecionar` já indica que a chamada desse método fará a seleção dos indivíduos e deverá retornar uma lista contendo os índices dos indivíduos escolhidos. A função `selecionar` terá como parâmetro de entrada o vetor *fitness*.

Já o método `selecao` vai retornar a nova população gerada via cruzamento:

```
    def selecao(self, n, fitness=None):
        """
        Retorna uma população de tamanho n,
        selecionada via método selecionar.
        """
        progenitores = array([self.selecionar(fitness)
                              for _ in range(n)])
        return self.populacao.populacao[progenitores]
```

Note que a única coisa que fazemos é rodar o método `selecionar` *n* vezes. Em seguida, retornamos apenas os elementos do vetor `populacao` que têm o índice dado pela variável `progenitores`.

Observe que esse não é um método abstrato, pois temos uma implementação concreta. Independentemente da estratégia de seleção adotada, o método `selecao` é geral o suficiente para ser usado diretamente em outros trechos de código que precisam usar a população selecionada. Esse método possui como parâmetro opcional o vetor *fitness* da população. A ideia é que, se seu valor for do tipo `None`, faremos com que o método concreto `selecionar` rode o método `avaliar` da classe `Populacao` cada vez que seja necessário fazer o processo de seleção. Caso seja passado o vetor `fitness`, estamos indicando para o algoritmo que a função de avaliação será executada apenas uma vez, para toda a população. Com isso conseguimos criar um código optimizado.

3.1 SELEÇÃO VIA ROLETA

Nesse método, que é similar a uma roleta, a seleção é proporcional ao valor obtido pela função de avaliação, também conhecido como *fitness* do indivíduo. A figura a seguir representa uma roleta para dez indivíduos. Cada divisão está organizada de acordo com o valor *fitness*, com largura indo do menor valor (indivíduos menos adaptados) para o maior (mais adaptabilidade). Mais adaptado indica maior probabilidade de se alcançar a melhor solução.

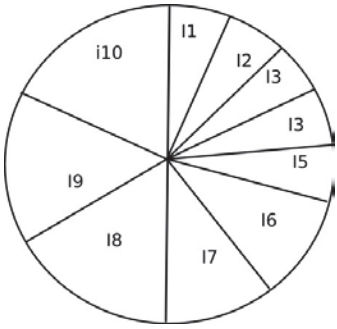


Figura 3.1: Roleta de Seleção

Se colocarmos a roleta para girar, teremos uma maior probabilidade de as áreas maiores pararem no topo da roda, indicando que aquele indivíduo foi selecionado para reprodução. Porém, nada impede que um indivíduo de menor valor *fitness* seja selecionado, mesmo tendo chances muito menores. A probabilidade de um progenitor ser escolhido é dada por:

$$p(i) = \frac{f(i)}{\sum_{j=1}^{n} f(j)}$$

Figura 3.2: Probabilidade de selecionar um indivíduo

sendo que a parte inferior da equação anterior,

$$\sum_{j=1}^{n} f(j)$$

Figura 3.3: Soma dos valores fitness

representa a soma dos valores *fitness* de todos os indivíduos.

A partir da classe mãe, vamos criar uma classe especialista, chamada `Roleta`, a qual implementará esse esquema de roleta a partir do método abstrato `selecionar`. Dentro da pasta `selecao` vamos criar o arquivo `roleta.py` que conterá a classe `Roleta`.

Começamos importando os módulos necessários (`random` para seleção aleatória, `array` para manipular vetores e a classe mãe `Selecao`):

```
from numpy.random import random
from numpy import array

from .selecao import Selecao
```

Em seguida criamos a classe e seu construtor:

```
class Roleta(Selecao):
    """
    Seleciona indivíduos para cruzamento usando
    roleta de seleção.
    Recebe como entrada:
        populacao - Objeto criado a partir da classe Populacao.
    """
    def __init__(self, populacao):
        super(Roleta, self).__init__(populacao)
```

A classe começa herdando os métodos da classe `Selecao`

(`class Roleta(Selecao)`). Observe que o construtor vai receber um objeto do tipo `Populacao` . Em seguida, passamos a variável `populacao` ao construtor da classe mãe, através do comando `super` (que inicializa a classe superior).

Agora podemos implementar o método `selecionar` , o qual usará a técnica da roleta:

```
def selecionar(self, fitness):
    """Roleta de seleção de indivíduos."""
    if fitness is None:
      fitness = self.populacao.avaliar()
    fmin = fitness.min()
    fitness = fitness - fmin
    total = fitness.sum()
    parada = total * (1.0 - random())
    parcial = 0
    i = 0
    while True:
        if i > fitness.size - 1:
            break
        parcial += fitness[i]
        if parcial >= parada:
            break
        i += 1
    return i - 1
```

A primeira coisa feita dentro do método é o cálculo da função *fitness* (`fitness = self.populacao.avaliar()`). Em seguida, encontramos o menor valor *fitness*, o qual é utilizado para normalizar todos os valores. Isso é feito para garantir que temos somente números positivos no vetor `fitness` , já que estamos convertendo esse número em dados de probabilidade de seleção de um indivíduo. Probabilidades negativas não possuem significado no processo de seleção, pois, ou um indivíduo será escolhido (chances positivas), ou não (probabilidade zero).

Como passo seguinte, calculamos a soma total dos valores

fitness da população para, em seguida, definirmos um valor de parada para a roleta (`parada = total * (1.0 - random())`). A posição de parada da roleta será definida de forma aleatória. Isso deixa a escolha do indivíduo para o acaso, mas sempre dando preferência aos que estão melhor adaptados ao espaço de busca do problema.

Usando o valor de parada, começamos a fazer a soma do menor valor para o maior. Quando o valor `parcial` alcança o valor de parada, o laço é interrompido e retornamos o indivíduo selecionado. Para dar chances aos indivíduos que ficaram com valor *fitness* igual a zero, no processo de normalização, retornamos `i-1` em vez de simplesmente o `i` .

O conteúdo completo do arquivo `roleta.py` será:

```
#!/usr/bin/env python3.6
# -*- Coding: UTF-8 -*-
"""
Roleta de Seleção de Indivíduos para cruzamento.

Programa sob licença GNU V.3.
Desenvolvido por: E. S. Pereira.
Versão 0.0.1.
"""

from numpy.random import random
from numpy import array

from .selecao import Selecao


class Roleta(Selecao):
    """
    Seleciona indivíduos para cruzamento usando
    roleta de seleção.
    Recebe como entrada:
        populacao - Objeto criado a partir da classe Populacao.
    """
```

```
    def __init__(self, populacao):
        super(Roleta, self).__init__(populacao)

    def selecionar(self, fitness):
        """Roleta de seleção de indivíduos."""
        if fitness is None:
          fitness = self.populacao.avaliar()
        fmin = fitness.min()
        fitness = fitness - fmin
        total = fitness.sum()
        parada = total * (1.0 - random())
        parcial = 0
        i = 0
        while True:
            if i > fitness.size - 1:
                break
            parcial += fitness[i]
            if parcial >= parada:
                break
            i += 1
        return i - 1
```

Vamos modificar o arquivo `main.py` para adicionar o método de seleção, acrescentando o seguinte comando:

```
from pygenec.selecao.roleta import Roleta


roleta = Roleta(populacao)
pop = roleta.selecao(10)

x, y = xy(pop)
```

Nesse caso, foi criado o objeto `roleta` e a variável `pop` contendo os indivíduos selecionados. Agora `x` e `y` são dados por essa subpopulação.

Expandindo um pouco mais o código:

```
cromossos_totais = 8
tamanho_populacao = 100
```

```
populacao = Populacao(avaliacao, cromossos_totais, tamanho_popula
cao)
populacao.gerar_populacao()

roleta = Roleta(populacao)
pop = roleta.selecao(10)

x, y = xy(pop)

fig = plt.figure(figsize=(100, 100))
ax = fig.add_subplot(111, projection="3d")
X, Y = mgrid[-3:3:30j, -3:3:30j]
Z = func(X,Y)

ax.plot_wireframe(X, Y, Z)
ax.scatter(x, y, func(x, y), s=50, c='red', marker='D')
plt.show()
```

Assim, estamos assumindo uma população de tamanho total 100, selecionando apenas 10% para o cruzamento. Na figura a seguir é possível observar os indivíduos que foram selecionados via roleta.

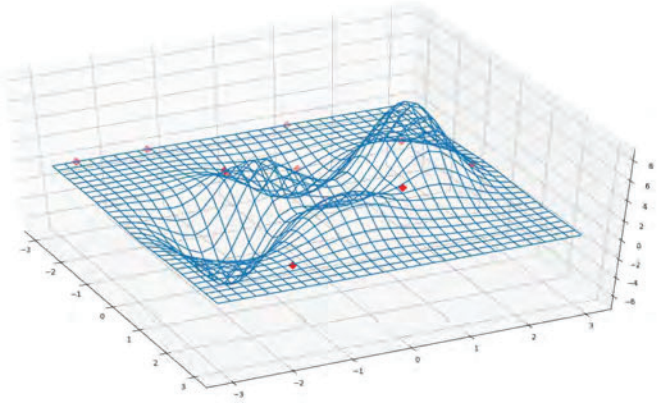


Figura 3.4: Os pontinhos vermelhos são indivíduos selecionados via Roleta.

3.2 SELEÇÃO POR CLASSIFICAÇÃO

O método por classificação é uma modificação do método da roleta. Nesse caso, em vez de utilizarmos diretamente o valor *fitness*, criamos uma classificação que vai de 1 até N. O indivíduo de menor valor *fitness* tem classificação 1 e o melhor tem classificação N.

O método de classificação garante a robustez da pressão de seleção, pois o mesmo define de forma clara quais são os melhores indivíduos, comparados um a um. Todavia, quando os valores *fitness* dos indivíduos da população são similares entre si (e.g. indivíduos `a` e `b` com valores *fitness* iguais a 0.010 e 0.011), o método começa a ter uma convergência mais lenta. Isso ocorre pelo fato de que a distribuição da probabilidade de seleção converge para um mesmo valor numérico.

A probabilidade de escolher um indivíduo nesse caso é dada por:

$$p(i) = class(i) / (n(n-1))$$

sendo `class(i)` a classificação do indivíduo `i` e `n` o maior valor de classificação para a população em análise.

No pacote `selecao` vamos criar o arquivo `classificacao.py` , no qual usaremos os seguintes módulos:

```
from numpy.random import random
from numpy import array, argsort

from .selecao import Selecao
```

A declaração da classe e seu construtor é dada por:

```
class Classificacao(Selecao):
    """
    Seleciona indivíduos para cruzamento usando
    Classificação.
    Recebe como entrada:
        populacao - Objeto criado a partir da classe Populacao.
    """
    def __init__(self, populacao):
        super(Classificacao, self).__init__(populacao)
```

Tal como a classe `Roleta`, essa classe recebe um objeto do tipo `populacao` em seu construtor.

Vamos agora implementar o método `selecionar` para esse caso específico:

```
def selecionar(self, fitness):
    """Roleta de seleção de indivíduos."""
    if fitness is None:
      fitness = self.populacao.avaliar()
    classificacao = argsort(fitness) + 1
    total = classificacao.sum()
    parada = total * random()
    parcial = 0
    i = 0
    while True:
        if i > fitness.size - 1:
            break
        parcial += classificacao[i]
        if parcial >= parada:
            break
        i += 1
    return i - 1
```

Na primeira linha, usamos o método `avaliar` para calcular o valor *fitness* da população. Em seguida, usamos a função `argsort` para obter o índice da ordenação crescente do *fitness* da população. Ao somar 1, estamos criando a classificação de 1 até N, sendo N o tamanho da população. Note que `total` representa a soma das classificações, a qual será usada para gerar um critério de parada.

Nesse caso, a variável `parcial` vai guardar a soma da classificação, interrompendo o laço caso ele alcance o critério de parada.

O código completo do módulo `classificacao.py` é apresentado a seguir:

```
#!/usr/bin/env python3.6
# -*- Coding: UTF-8 -*-
"""
Roleta de Seleção de Indivíduos para cruzamento, classificação.

Programa sob licença GNU V.3.
Desenvolvido por: E. S. Pereira.
Versão 0.0.1.
"""

from numpy.random import random
from numpy import array, argsort

from .selecao import Selecao

class Classificacao(Selecao):
    """
    Seleciona indivíduos para cruzamento usando
    Classificação.
    Recebe como entrada:
        populacao - Objeto criado a partir da classe Populacao.
    """
    def __init__(self, populacao):
        super(Classificacao, self).__init__(populacao)

    def selecionar(self, fitness):
        """Roleta de seleção de indivíduos."""
        if fitness is None:
          fitness = self.populacao.avaliar()
        classificacao = argsort(fitness) + 1
        total = classificacao.sum()
        parada = total * random()
        parcial = 0
        i = 0
        while True:
```

```
            if i > fitness.size - 1:
                break
            parcial += classificacao[i]
            if parcial >= parada:
                break
            i += 1
        return i - 1
```

Para ver os resultados mudamos o programa `main.py` :

```
from pygenec.selecao.classificacao import Classificacao

classificacao = Classificacao(populacao)
pop = classificacao.selecao(10)

x, y = xy(pop)
```

Na figura a seguir, os pontos em vermelho representam as possíveis soluções encontradas via classificação.

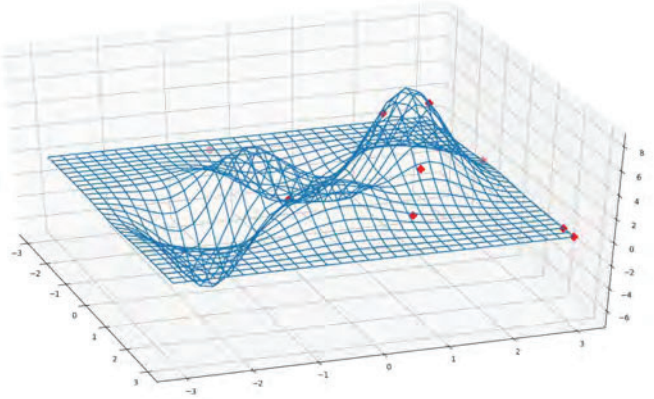


Figura 3.5: Os pontinhos vermelhos são indivíduos selecionados via classificação.

3.3 SELEÇÃO POR TORNEIO

Na seleção por torneio, um subgrupo é selecionado aleatoriamente, a partir da população maior. Em seguida, esse

subgrupo compete entre si, tal que o ganhador da rodada é aquele que tiver o maior valor *fitness*. Os ganhadores farão parte do grupo que será usado para reprodução. A vantagem desse método é que ele dá uma chance igual para que todos os indivíduos tenham oportunidade de se reproduzir, o que acaba levando à preservação da diversidade genética das próximas gerações.

A desvantagem é que selecionar soluções não muito boas para reprodução reduzirá a velocidade de convergência para a solução. Mas como estamos levando em conta o fator acaso, combinações de soluções não muito boas ajudam a ampliar o espaço de busca, reduzindo as chances de o método ficar preso em uma solução local, ajudando a encontrar uma solução global para um dado problema.

No pacote `selecao` vamos criar o arquivo `torneio.py`. Para esse módulo, vamos importar as seguintes funções:

```
from numpy.random import choice
from numpy import array, where
from .selecao import Selecao
```

A declaração da classe e seu construtor será:

```
class Torneio(Selecao):
    """
    Seleciona indivíduos para cruzamento usando
    Torneio.
    Recebe como entrada:
        populacao - Objeto criado a partir da classe Populacao.
    """
    def __init__(self, populacao, tamanho=10):
        super(Torneio, self).__init__(populacao)
        self.tamanho = tamanho
```

Da mesma forma que nos outros casos, a classe recebe um objeto do tipo `Populacao`, mas aqui ele passa a ter uma entrada

padrão, chamada `tamanho`, com valor igual a 10. O atributo `tamanho` representa o tamanho do subgrupo que vai competir entre si a cada rodada de torneio.

O método `selecionar` será:

```
def selecionar(self, fitness):
    """Retorna o indivíduo campeão da rodada."""
    if fitness is None:
      fitness = self.populacao.avaliar()
    grupo = choice(fitness, size=self.tamanho)
    campeao = grupo.max()
    i = where(fitness == campeao)[0][0]
    return i
```

Inicialmente, usamos o método `avaliar` para obter os valores *fitness* da população. Em seguida, usamos a função `choice`, que vai retornar um subgrupo, gerado aleatoriamente, a partir do vetor `fitness`. O tamanho desse subgrupo é passado pelo parâmetro `size`, que será dado pelo atributo `tamanho`, que por padrão tem valor igual a 10.

Em seguida, escolhemos o campeão, que é o maior valor *fitness* dentro do subgrupo (`campeao = grupo.max()`). Para achar o índice que corresponde ao indivíduo campeão na população usamos a função `where`. Essa função recebe uma comparação e retorna as posições do primeiro vetor (no caso, o vetor `fitness`) que satisfazem a comparação (`fitness == campeao`). O `where` retorna uma tupla. Se o vetor a ser analisado for equivalente a uma matriz, o primeiro termo da tupla será um vetor contendo as linhas da matriz que satisfazem a condição de comparação, já o segundo elemento representará as colunas. Caso o vetor seja formado apenas por linhas, o segundo elemento da tupla será vazio.

O valor de `i` será dado pelo primeiro indivíduo, no subgrupo,

que satisfaz a condição de ter o maior valor *fitness*.

O código completo do módulo `torneio.py` é apresentado a seguir:

```
#!/usr/bin/env python3.6
# -*- Coding: UTF-8 -*-
"""
Roleta de Seleção de Indivíduos para cruzamento.

Programa sob licença GNU V.3.
Desenvolvido por: E. S. Pereira.
Versão 0.0.1.
"""

from numpy.random import choice
from numpy import array, where
from .selecao import Selecao


class Torneio(Selecao):
    """
    Seleciona indivíduos para cruzamento usando
    Torneio.
    Recebe como entrada:
        populacao - Objeto criado a partir da classe Populacao.
    """
    def __init__(self, populacao, tamanho=10):
        super(Torneio, self).__init__(populacao)
        self.tamanho = tamanho

    def selecionar(self, fitness):
        """Retorna o indivíduo campeão da rodada."""
        if fitness is None:
          fitness = self.populacao.avaliar()
        grupo = choice(fitness, size=self.tamanho)
        campeao = grupo.max()
        i = where(fitness == campeao)[0][0]
        return i
```

Para visualizar os resultados, modificamos o programa `main.py`, importando essa nova classe:

```
from pygenec.selecao.torneio import Torneio
```

Em seguida, instanciando o objeto `torneio` :

```
cromossos_totais = 8
tamanho_populacao = 100

populacao = Populacao(avaliacao, cromossos_totais, tamanho_popula
cao)
populacao.gerar_populacao()

classificacao = Torneio(populacao, tamanho=10)
pop = classificacao.selecao(10)

x, y = xy(pop)
```

Na próxima figura vemos em vermelho as soluções que representam os indivíduos selecionados via torneio.

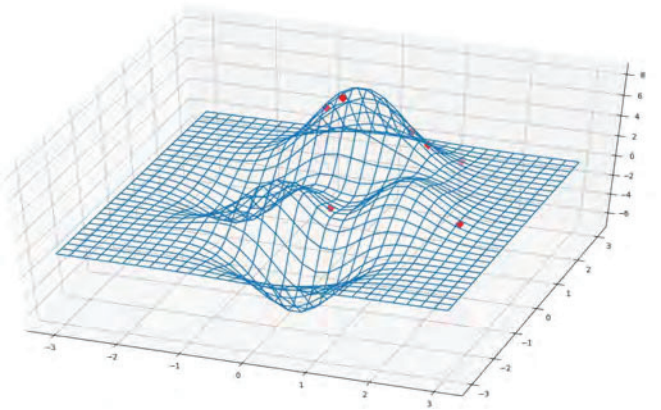


Figura 3.6: Os pontinhos vermelhos são indivíduos selecionados via Torneio.

A estrutura atual do projeto ficou da seguinte forma:

```
pygenec\
 |
 |-pygenec\
           |-__init__.py
           |-populacao.py
```

```
        |-selecao\
                |-__init__.py
                |-selecao.py
                |-roleta.py
                |-classificacao.py
                |-torneio.py
|-main.py
```

Conclusão

Neste capítulo foram apresentados alguns dos principais métodos de seleção de indivíduos: Roleta, Classificação e Torneio. Existem outros, tais como (Jebari e Madiafi, 2013): Amostra Estocástica Universal, Seleção por Classificação Exponencial e Seleção por Truncamento. A escolha de qual método usar vai depender de cada problema.

Vimos que a escolha interfere na velocidade de convergência para uma possível solução. Contudo, é preciso tomar cuidado para que o método não fique preso em uma solução local. Como regra geral, temos que, para algoritmos genéticos, somente através da experimentação e teste é que se consegue efetivamente descobrir qual será a seleção mais indicado para o problema que se quer resolver.

No próximo capítulo veremos alguns dos principais métodos de cruzamento.

Capítulo 4

CRUZAMENTO

Após a seleção dos indivíduos mais adaptados, o próximo passo é o de realizar o cruzamento. O cruzamento visa combinar qualidades genéticas, gerando uma nova população que seja mais resistente à pressão de seleção, definindo quais serão os genes a serem passados para a próxima geração. Essa combinação conduz o processo de evolução na busca da melhor solução possível de um dado problema que se quer resolver.

Vários operadores de cruzamento foram criados com o objetivo de se obter uma solução ótima o mais rápido possível. A escolha adequada de um operador evita uma convergência prematura, ou seja, reduz as chances de o algoritmo ficar preso em uma solução local.

Um estudo interessante sobre o papel da escolha do operador de cruzamento é feito por Umbarkar e Sheth (2015). Esses autores descrevem que os operadores de cruzamento podem ser classificados em três categorias: cruzamento padrão, cruzamento binário e cruzamento real/árvore. Neste livro vamos abordar o cruzamento padrão, em particular os seguintes operadores de cruzamento: de um ponto; de k-pontos; embaralhamento.

Para começar, dentro do `pygenec`, no pacote chamado

`cruzamento`, vamos criar o arquivo `cruzamento.py`. Tal como para o operador de seleção, aqui nós vamos criar uma classe base, chamada `Cruzamento`, a qual será utilizada como mãe para os operadores desse pacote.

A nova estrutura do projeto será a seguinte:

```
pygenec\
 |
 |-pygenec\
           |-__init__.py
           |-populacao.py
           |-selecao\
                    |-__init__.py
                    |-roleta.py
                    |-classificacao.py
                    |-torneio.py
           |-cruzamento\
                    |- __init__.py
                    |- cruzamento.py
 |-main.py
```

Vamos utilizar aqui as seguintes chamadas do numpy:

```
from numpy import array
from numpy.random import randint, random
```

O construtor da classe `Cruzamento`, que receberá como parâmetro o tamanho da população a ser gerada via cruzamento, será:

```
class Cruzamento:
    """
    Classe abstrata representando o cruzamento.

    Entrada:
        tamanho_populacao - Tamanho final da população resultante
.
    """
    def __init__(self, tamanho_populacao):
        self.tamanho_populacao = tamanho_populacao
```

Essa classe terá dois métodos `cruzamento`, o qual deverá ser implementado em cada nova classe, e o `descendentes`, responsável por retornar a nova população.

O `cruzamento` será:

```
def cruzamento(self, progenitor1, progenitor2):
    raise NotImplementedError("A ser implementado")
```

A geração da nova população ficará a cargo do método `descendentes`:

```
def descendentes(self, subpopulacao, pcruz):
      """
      Retorna uma nova população de tamanho tamanho_populacao
      através do cruzamento.

      Entrada:
          subpopulacao - Vetor contendo indivíduos para serem sel
ecionados
                         para cruzamento.
          pcruz - probabilidade de cruzamento entre dois indivídu
os
                  selecionados.
      """
      nova_populacao = []
      npop = len(subpopulacao)

      while len(nova_populacao) < self.tamanho_populacao:
          i = randint(0, npop - 1)
          j = randint(0, npop - 1)
          while j == i:
              j = randint(0, npop - 1)

          cruzar = random()
          if cruzar < pcruz:
              desc1, desc2 = self.cruzamento(subpopulacao[i], sub
populacao[j])
              nova_populacao.append(desc1)
              if len(nova_populacao) < self.tamanho_populacao:
                  nova_populacao.append(desc2)
```

```
        return array(nova_populacao)
```

Nesse método, inicialmente, criamos uma lista vazia para guardar os indivíduos da nova população. Como são gerados sempre dois descendentes, executamos o laço de repetição até que o tamanho da nova população seja menor que o valor do atributo `tamanho_populacao`. Dois progenitores são escolhidos aleatoriamente dentro da subpopulação (`i = randint(0, npop - 1)` e `j = randint(0, npop - 1)`). A variável `cruzar` mudará aleatoriamente a cada rodada, sendo usada para considerar se os dois indivíduos selecionados serão efetivamente utilizados, ou não, no processo de cruzamento. Se o valor de `cruzar` for menor que `pcruz` então dois descendentes são gerados e adicionados à nova população. Ao final o método retorna um vetor contendo a nova população.

Vale lembrar que na natureza o cruzamento não vai ocorrer simplesmente pelo encontro entre um macho e uma fêmea. Em muitos casos, o ritual de acasalamento não termina com a união do casal. Essa é a relação entre as variáveis `pcruz` e `cruzar`. Se `pcruz` for elevado, maiores serão as chances de o casal escolhido se cruzar ao se encontrarem pela primeira vez. Sabemos que isso não é absolutamente verdade na natureza. Porém, um valor muito baixo de `pcruz` fará com que o processo de cruzamento fique mais lento. Cabe ao desenvolvedor analisar o problema e definir qual deverá ser o parâmetro ideal.

Vamos criar uma extensão da classe `Exception` para gerar uma exceção customizada, a qual utilizaremos para comparar o tamanho de dois progenitores a serem usados no cruzamento.

```
class NoCompatibleIndividualSize(Exception):
    pass
```

Isso será útil para garantir que o cruzamento só ocorra entre indivíduos que tenha o mesmo perfil de carga genética.

O código completo do módulo será:

```
from numpy import array
from numpy.random import randint, random

class NoCompatibleIndividualSize(Exception):
    pass

class Cruzamento:
    """
    Classe abstrata representando o cruzamento.

    Entrada:
        tamanho_populacao - Tamanho final da população resultante
.
    """
    def __init__(self, tamanho_populacao):
        self.tamanho_populacao = tamanho_populacao

    def cruzamento(self, progenitor1, progenitor2):
        raise NotImplementedError("A ser implementado")

    def descendentes(self, subpopulacao, pcruz):
          """
          Retorna uma nova população de tamanho tamanho_populacao
          através do cruzamento.

          Entrada:
              subpopulacao - Vetor contendo indivíduos para serem
 selecionados
                             para cruzamento.
              pcruz - probabilidade de cruzamento entre dois indi
víduos
                      selecionados.
          """
          nova_populacao = []
          npop = len(subpopulacao)

          while len(nova_populacao) < self.tamanho_populacao:
```

```
                i = randint(0, npop - 1)
                j = randint(0, npop - 1)
                while j == i:
                    j = randint(0, npop - 1)

                cruzar = random()
                if cruzar < pcruz:
                    desc1, desc2 = self.cruzamento(subpopulacao[i],
 subpopulacao[j])
                    nova_populacao.append(desc1)
                    if len(nova_populacao) < self.tamanho_populacao
:
                        nova_populacao.append(desc2)

            return array(nova_populacao)
```

4.1 CRUZAMENTO DE UM PONTO

O primeiro operador que veremos é o cruzamento de um ponto. Nesse caso, dois progenitores são selecionados e um ponto de corte é escolhido aleatoriamente. Em seguida, as partes são cruzadas, gerando dois descendentes. Na figura a seguir é representado esse processo.

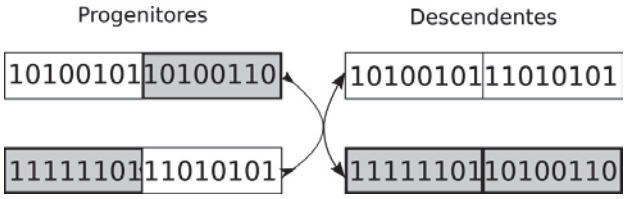


Figura 4.1: Cruzamento em um ponto.

Dentro do pacote `cruzamento` vamos criar o arquivo `umponto.py` . Para esse módulo importaremos as seguintes funções e classes:

```
from numpy.random import randint
from numpy import array
```

```
from .cruzamento import Cruzamento, NoCompatibleIndividualSize
```

A classe `Umponto` receberá em seu construtor o tamanho da população a ser gerada via cruzamento:

```
class UmPonto(Cruzamento):
    """
    Gerador de população via cruzamento usando o operador de um p
onto.

    Entrada:
        tamanho_populacao - Tamanho final da população resultante
.
    """
    def __init__(self, tamanho_populacao):
        super(UmPonto, self).__init__(tamanho_populacao)
```

O método `cruzamento` receberá dois progenitores e retornará dois descendentes:

```
def cruzamento(selfself, progenitor1, progenitor2):
    """
    Cruzamento de dois indivíduos via um ponto.

    Entrada:
        ind1 - Primeiro indivíduo
        ind2 - Segundo indivíduo
    O tamanho de ambos os indivíduos deve ser igual, do contrário
 um erro
    será levantado.
    """
    n1 = len(progenitor1)
    n2 = len(progenitor2)

    if n1 != n2:
        msg = "Tamanho ind1 {0} diferente de ind2 {1}".format(n1,
 n2)
        raise NoCompatibleIndividualSize(msg)

    ponto = randint(1, n1 - 1)
    desc1 = progenitor1.copy()
    desc2 = progenitor2.copy()
```

```
    desc1[ponto:] = progenitor2[ponto:]
    desc2[ponto:] = progenitor1[ponto:]
    return desc1, desc2
```

Primeiramente, o método verifica se os dois progenitores têm tamanhos compatíveis. Caso isso não ocorra, um erro é levantado. A variável `ponto` armazena o ponto de corte, o qual será definido aleatoriamente dentro do intervalo de 1 até o último ponto do vetor progenitor (`ponto = randint(1, n1 - 1)`). Os descendentes serão cópias dos progenitores (`desc1 = progenitor1.copy()`).

O cruzamento é feito modificando a carga genética a partir do ponto de corte. Por exemplo, o `desc1` foi construído inicialmente como sendo uma cópia do `progenitor1` . Em seguida, a primeira parte de seu código genético é substituída pela primeira carga genética do `progenitor2` (`desc1[ponto:] = progenitor2[ponto:]`). A mesma coisa é feita com o `desc2` .

O código completo contido no arquivo `umponto.py` é apresentado a seguir:

```
#!/usr/bin/env python3.6
# -*- Coding: UTF-8 -*-
"""
Cruzamento por um ponto.

Programa sob licença GNU V.3.
Desenvolvido por: E. S. Pereira.
Versão 0.0.1.
"""

from numpy.random import randint
from numpy import array

from .cruzamento import Cruzamento, NoCompatibleIndividualSize
```

```
class UmPonto(Cruzamento):
    """
    Gerador de população via cruzamento usando o operador de um p
onto.

    Entrada:
        tamanho_populacao - Tamanho final da população resultante
.
    """
    def __init__(self, tamanho_populacao):
        super(UmPonto, self).__init__(tamanho_populacao)

    def cruzamento(selfself, progenitor1, progenitor2):
        """
        Cruzamento de dois indivíduos via um ponto.

        Entrada:
            ind1 - Primeiro indivíduo
            ind2 - Segundo indivíduo
        O tamanho de ambos os indivíduos deve ser igual, do contr
ário um erro
        será levantado.
        """
        n1 = len(progenitor1)
        n2 = len(progenitor2)

        if n1 != n2:
            msg = "Tamanho ind1 {0} diferente de ind2 {1}".format
(n1, n2)
            raise NoCompatibleIndividualSize(msg)

        ponto = randint(1, n1 - 1)
        desc1 = progenitor1.copy()
        desc2 = progenitor2.copy()

        desc1[ponto:] = progenitor2[ponto:]
        desc2[ponto:] = progenitor1[ponto:]
        return desc1, desc2
```

4.2 CRUZAMENTO EM K-PONTOS

No cruzamento de k-pontos, dois progenitores selecionados

são subdivididos em k partes. A escolha do total de partes a ser considerada é determinada aleatoriamente, variando de `0` a `n-1` pontos, sendo `n` o tamanho total de um dado indivíduo da população. Dois descendentes são gerados combinando os progenitores nos pontos de cruzamento.

Na figura a seguir é exemplificado cruzamento de k-pontos. O descendente superior é formado pelo primeiro e terceiro ponto de corte do progenitor superior, combinados com o conjunto de genes formados pelo segundo e quarto ponto de corte do progenitor inferior. O segundo descendente é formado pelos pontos de corte opostos, ou seja, primeiro e terceiro ponto de corte do progenitor inferior e segundo e quarto ponto de corte do progenitor superior.

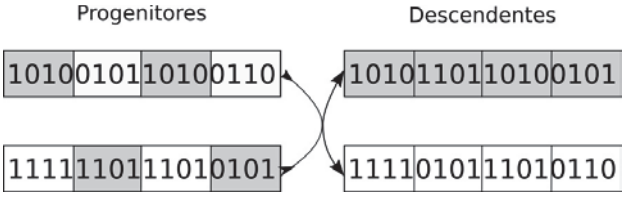


Figura 4.2: Cruzamento em k-pontos.

Dentro do pacote `cruzamento` vamos criar o arquivo `kpontos.py`. Para esse módulo vamos importar as seguintes funções e classes:

```
from numpy.random import randint, random
from numpy import array

from .cruzamento import Cruzamento, NoCompatibleIndividualSize
```

A declaração da classe e seu construtor será:

```
class KPontos(Cruzamento):
```

```
    """
    Gerador de população via cruzamento usando o operador k-ponto
s.

    Entrada:
        tamanho_populacao - Tamanho final da população resultante
.
    """
    def __init__(self, tamanho_populacao):
        super(KPontos, self).__init__(tamanho_populacao)
```

A classe `KPontos` receberá no construtor o tamanho da população a ser gerada via cruzamento.

O método `cruzamento` realizará a operação de cruzamento, recebendo como entrada dois progenitores previamente selecionados:

```
def cruzamento(self, progenitor1, progenitor2):
    """
    Cruzamento via k-pontos de dois indivíduos.

    Entrada:
        ind1 - Primeiro indivíduo
        ind2 - Segundo indivíduo
    O tamanho de ambos os indivíduos deve ser igual, do contrário
 um erro
    será levantado.
    """
    n1 = len(progenitor1)
    n2 = len(progenitor2)

    if n1 != n2:
        msg = "Tamanho ind1 {0} diferente de ind2 {1}".format(n1,
 n2)
        raise NoCompatibleIndividualSize(msg)

    desc1 = progenitor1.copy()
    desc2 = progenitor2.copy()

    kp = randint(1, n1-2)

    k = []
```

```
    while len(k) <= kp:
        p = randint(1, n1-1)
        if p not in k:
            k.append(p)
    k.sort()

    troca = randint(0, 1)

    for i in range(kp):
        if troca == 0:
            troca = 1
        else:
            troca = 0
            desc1[k[i]:k[i + 1]] = progenitor2[k[i]:k[i + 1]]
            desc2[k[i]:k[i + 1]] = progenitor1[k[i]:k[i + 1]]

    return desc1, desc2
```

A primeira coisa que o método faz é verificar se os dois progenitores têm tamanhos compatíveis. Caso isso não seja verdade, um erro é levantado. Após a validação dos progenitores, os descendentes são inicialmente gerados a partir de cópias dos progenitores.

O comando `kp = randint(1, n1-2)` será usado para gerar os k-pontos de cruzamento. Os pontos são armazenados na lista `k`. O seguinte trecho criará uma lista contendo efetivamente os pontos a serem considerados no cruzamento:

```
while len(k) <= kp:
    p = randint(1, n1-1)
    if p not in k:
        k.append(p)
k.sort()
```

Note que um valor `p` é produzido aleatoriamente dentro do intervalo `1` até `n1-1`. Se o novo ponto não estiver na lista, então ele será incluído. Usando o método `sort` da lista, fazemos a

ordenação crescente dos pontos de cruzamento.

A variável `troca` é utilizada para indicar se um trecho de código genético deve ser trocado ou não. Observe que, para cada trecho dos k-pontos, uma troca é feita e outra não:

```
if troca == 0:
    troca = 1
else:
    troca = 0
    desc1[k[i]:k[i + 1]] = progenitor2[k[i]:k[i + 1]]
    desc2[k[i]:k[i + 1]] = progenitor1[k[i]:k[i + 1]]
```

No próximo código é apresentado o programa completo contido no arquivo `kpontos.py` :

```
#!/usr/bin/env python3.6
# -*- Coding: UTF-8 -*-
"""
Cruzamento por k-pontos.

Programa sob licença GNU V.3.
Desenvolvido por: E. S. Pereira.
Versão 0.0.1.
"""

from numpy.random import randint, random
from numpy import array

from .cruzamento import Cruzamento, NoCompatibleIndividualSize

class KPontos(Cruzamento):
    """
    Gerador de população via cruzamento usando o operador k-ponto
s.

    Entrada:
        tamanho_populacao - Tamanho final da população resultante
.
    """
    def __init__(self, tamanho_populacao):
        super(KPontos, self).__init__(tamanho_populacao)
```

```
    def cruzamento(self, progenitor1, progenitor2):
        """
        Cruzamento via k-pontos de dois indivíduos.

        Entrada:
            ind1 - Primeiro indivíduo
            ind2 - Segundo indivíduo
        O tamanho de ambos os indivíduos deve ser igual, do contr
ário um erro
        será levantado.
        """
        n1 = len(progenitor1)
        n2 = len(progenitor2)

        if n1 != n2:
            msg = "Tamanho ind1 {0} diferente de ind2 {1}".format
(n1, n2)
            raise NoCompatibleIndividualSize(msg)

        desc1 = progenitor1.copy()
        desc2 = progenitor2.copy()

        kp = randint(1, n1-2)

        k = []

        while len(k) <= kp:
            p = randint(1, n1-1)
            if p not in k:
                k.append(p)
        k.sort()

        troca = randint(0, 1)

        for i in range(kp):
            if troca == 0:
                troca = 1
            else:
                troca = 0
                desc1[k[i]:k[i + 1]] = progenitor2[k[i]:k[i + 1]]
                desc2[k[i]:k[i + 1]] = progenitor1[k[i]:k[i + 1]]

        return desc1, desc2
```

Vamos modificar o arquivo `main.py` para adicionar esse novo operador:

```
from pygenec.cruzamento.kpontos import KPontos
```

Para criar um novo objeto de cruzamento modificamos o seguinte trecho de código:

```
cromossos_totais = 8
tamanho_populacao = 100

populacao = Populacao(avaliacao, cromossos_totais, tamanho_popula
cao)
populacao.gerar_populacao()

classificacao = Torneio(populacao, tamanho=10)
subpopulacao = classificacao.selecao(10)

kpontos = KPontos(tamanho_populacao)
pop = kpontos.descendentes(subpopulacao, pcruz=0.5)

x, y = xy(pop)
```

O gráfico a seguir exibe as soluções obtido usando os indivíduos gerados por cruzamento através da classe `KPontos` .

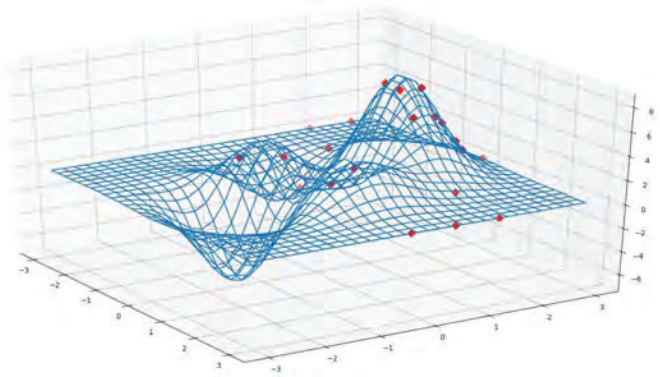


Figura 4.3: Cruzamento em k-pontos.

4.3 EMBARALHAMENTO

No cruzamento com embaralhamento, primeiramente cada progenitor selecionado é previamente embaralhado. Em seguida, usando a técnica de cruzamento de um ponto, dois novos descendentes são gerados. Por fim, os descendentes gerados sofrem embaralhamento reverso (desembaralhado ou reorganização), ou seja, o código genético é reorganizado obedecendo o sentido contrário do embaralhamento. A vantagem dessa técnica é que ela remove o viés gerado pela escolha do ponto de corte, pois a cada rodada os indivíduos são embaralhados de forma diferente (ver http://www.cse.unsw.edu.au/~cs9417ml/GA2/crossover_alternate.html/).

Nas próximas figuras o conceito de embaralhamento é apresentado.

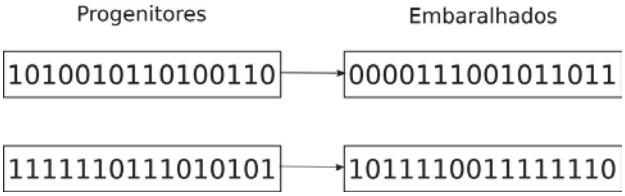


Figura 4.4: Embaralhamento dos progenitores.

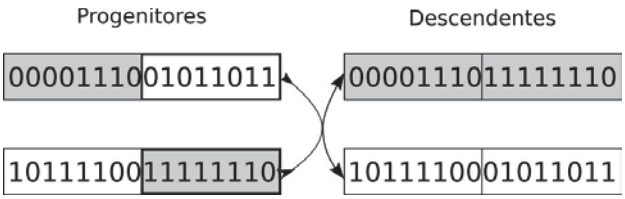


Figura 4.5: Cruzamento via um ponto.

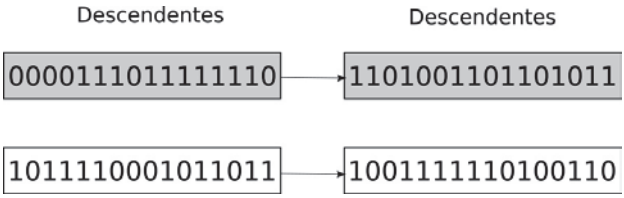


Figura 4.6: Embaralhamento reverso.

Dentro do pacote `cruzamento` vamos criar o arquivo `embaralhamento.py` . Para esse módulo importaremos o seguinte:

```
from numpy.random import shuffle, randint
from .cruzamento import Cruzamento, NoCompatibleIndividualSize
```

A implementação do método `cruzamento` será:

```
def cruzamento(selfself, progenitor1, progenitor2):
    """
    Cruzamento de dois indivíduos via embaralhamento um ponto.

    Entrada:
        ind1 - Primeiro indivíduo
        ind2 - Segundo indivíduo
    O tamanho de ambos os indivíduos deve ser igual, do contrário
 um erro
    será levantado.
    """
    n1 = len(progenitor1)
    n2 = len(progenitor2)

    if n1 != n2:
        msg = "Tamanho ind1 {0} diferente de ind2 {1}".format(n1,
 n2)
        raise NoCompatibleIndividualSize(msg)

    order = list(range(n1))
    shuffle(order)

    ponto = randint(1, n1 - 1)
    desc1 = progenitor1.copy()
    desc2 = progenitor2.copy()
```

```
    desc1[:] = desc1[order]
    desc2[:] = desc2[order]

    desc1[ponto:] = desc2[ponto:]
    desc2[ponto:] = desc1[ponto:]

    tmp1 = desc1.copy()
    tmp2 = desc2.copy()

    for i, j in enumerate(order):
        desc1[j] = tmp1[i]
        desc2[j] = tmp2[i]

    return desc1, desc2
```

Primeiramente, uma lista representando o índice dos progenitores é armazenada na variável `order`. Usando a função `shuffle`, essa lista é embaralhada. O ponto de corte para o cruzamento é gerado e armazenado na variável `ponto`. Uma cópia dos progenitores é usada para inicializar os descendentes.

Em seguida, é realizado o embaralhamento, usando a lista de índices `order`, que foi embaralhada previamente. Então ocorre o cruzamento de um ponto dos dois descentes de forma simultânea (`desc1[ponto:]=desc2[ponto:]` e `desc2[ponto:]=desc1[ponto:]`). Para desfazer o embaralhamento, nos descendentes, usamos a função `enumerate` que retorna a posição e os valores da lista enumeradamente.

A seguir está o código completo do módulo:

```
#!/usr/bin/env python3.6
# -*- Coding: UTF-8 -*-
"""
Cruzamento via embaralhamento.

Programa sob licença GNU V.3.
Desenvolvido por: E. S. Pereira.
```

```
Versão 0.0.1.
"""

from numpy.random import shuffle, randint
from .cruzamento import Cruzamento, NoCompatibleIndividualSize


class Embaralhamento(Cruzamento):
    """
    Gerador de população via embaralhamento e cruzamento de um po
nto.

    Entrada:
        tamanho_populacao - Tamanho final da população resultante
.
    """
    def __init__(self, tamanho_populacao):
        super(Embaralhamento, self).__init__(tamanho_populacao)

    def cruzamento(self, progenitor1, progenitor2):
        """
        Cruzamento de dois indivíduos via embaralhamento um ponto
.

        Entrada:
            ind1 - Primeiro indivíduo
            ind2 - Segundo indivíduo
        O tamanho de ambos os indivíduos deve ser igual, do contr
ário um erro
        será levantado.
        """
        n1 = len(progenitor1)
        n2 = len(progenitor2)

        if n1 != n2:
            msg = "Tamanho ind1 {0} diferente de ind2 {1}".format
(n1, n2)
            raise NoCompatibleIndividualSize(msg)

        order = list(range(n1))
        shuffle(order)

        ponto = randint(1, n1 - 1)
        desc1 = progenitor1.copy()
        desc2 = progenitor2.copy()
```

```
        desc1[:] = desc1[order]
        desc2[:] = desc2[order]

        desc1[ponto:], desc2[ponto:] =  desc2[ponto:], desc1[pont
o:]

        tmp1 = desc1.copy()
        tmp2 = desc2.copy()

        for i, j in enumerate(order):
            desc1[j] = tmp1[i]
            desc2[j] = tmp2[i]

        return desc1, desc2
```

Novamente modificamos o programa `main.py` para adicionar esse novo operador:

```
from pygenec.cruzamento.embaralhamento import Embaralhamento
```

E também fazemos a seguinte modificação:

```
cromossos_totais = 8
tamanho_populacao = 100

populacao = Populacao(avaliacao, cromossos_totais, tamanho_popula
cao)
populacao.gerar_populacao()

classificacao = Torneio(populacao, tamanho=10)
subpopulacao = classificacao.selecao(10)

embaralhamento = Embaralhamento(tamanho_populacao)
pop = embaralhamento.descendentes(subpopulacao, pcruz=0.5)

x, y = xy(pop)
```

A seguinte figura representa o resultado obtido via cruzamento com embaralhamento.

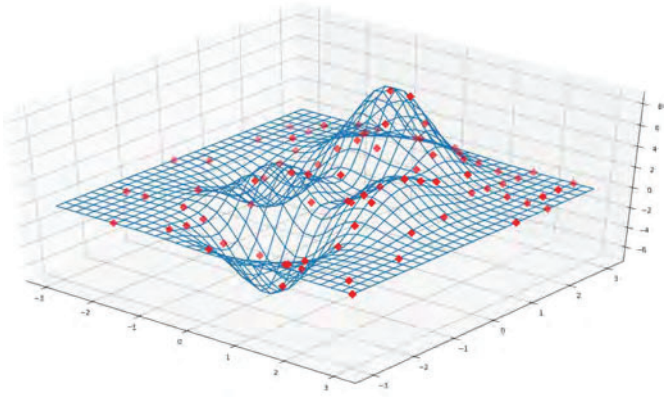


Figura 4.7: Embaralhamento reverso.

Conclusão

Neste capítulo foram abordados os operadores de cruzamento padrão, utilizados para combinar possíveis soluções de um dado problema. Foram implementados os operadores padrão de um ponto, k-pontos e por embaralhamento. O motivo de se criar diferentes operadores de cruzamento deriva da busca por métodos que levem a convergência mais rápida para uma solução global ótima do problema que se quer resolver.

No próximo capítulo vamos abordar o operador de mutação.

Capítulo 5

MUTAÇÃO

De acordo com o livro de Neto et. al. (2016), o principal papel da mutação é o de proteger a busca genética de uma perda prematura de material genético bom, sendo que essa perda pode ocorrer durante o processo de seleção e cruzamento. Além disso, a modificação aleatória de genes, que ajuda a manter a diversidade genética da população, previne que a busca fique presa a uma solução local.

No contexto da teoria da evolução, alguns casos de mutações podem trazer benefícios, gerando indivíduos com novas características e mais adaptados ao ambiente. Por outro lado, sabemos que nem sempre a mutação é benéfica. Se a probabilidade de ocorrência de transformação genética for elevada, a busca pela solução do problema se torna meramente aleatória, não representando benefício real para se alcançar a melhor solução possível.

Da mesma forma que para os outros operadores vistos anteriormente, existe um grande número de operadores de mutação. Neste capítulo veremos três dos operadores mais utilizados: Mutação *flip*, Mutação de dupla troca e Mutação de sequência reversa.

No nosso código, vamos criar o pacote `mutacao` e o módulo `mutacao.py`, deixando o projeto com a seguinte estrutura:

```
pygenec\
 |
 |-pygenec\
           |-__init__.py
           |-populacao.py
           |-selecao\
                    |-__init__.py
                    |-roleta.py
                    |-classificacao.py
                    |-torneio.py
           |-cruzamento\
                    |-__init__.py
                    |-cruzamento.py
                    |-embaralhamento.py
                    |-kpontos.py
                    |-umponto.py
            |-mutacao\
                    |-__init__.py
                    |-mutacao.py
 |-main.py
```

A classe base será chamada `Mutacao`, contendo o método abstrato `mutacao`. Vamos importar as seguintes funções:

```
from numpy.random import randint, random
from numpy import array
```

A declaração da classe e seu construtor será:

```
class Mutacao:
    """
    Classe base para operadores de mutação:
    Entrada:
        pmut - probabilidade de ocorrer uma mutação.
    """

    def __init__(self, pmut):
        self.pmut = pmut
        self._populacao = None
        self.npop = None
```

```
        self.ngen = None
```

O atributo `_populacao` será usado como apontador para o vetor contendo a população que sofrerá mutação.

Público, privado e protegido em Python:

Em Python, todo método ou atributo pode ser acessado. Contudo, existe um padrão de nome que dificulta o acesso direto, permitindo criar o efeito de privado e protegido. Um método, ou atributo, privado é aquele que deve ser usado apenas pela classe mãe.

Por convenção, nomes iniciados com um *underline* (_) serão assumidos como sendo do tipo privado. Já nomes com dois *underlines* (__) serão métodos do tipo protegido, e podem ser diretamente pelos filhos da classe mãe. (Para saber mais, recomenda-se ver capítulo 5 de Pereira (2018) sobre Programação Orientada a Objetos em Python.).

Usaremos o `property` para criar o *getter* e o *setter* do atributo `_populacao`.

```
def _set_populacao(self, populacao):
    self._populacao = populacao
    self.npop = self._populacao.shape[0]
    self.ngen = self._populacao.shape[1]

def _get_populacao(self):
    return self._populacao
```

Nesse momento também definimos os atributos `npop` e `ngen` que representam, respectivamente, o tamanho da população

e o número de genes contido nos indivíduos. Ao final da classe adicionamos o comando:

```
populacao = property(_get_populacao, _set_populacao)
```

Assim, ao utilizarmos o atributo `populacao` estaremos na verdade usando os métodos `_get_populacao` e `_set_populacao`. O método `selecao` vai escolher os indivíduos da população que sofrerão mutação, levando em conta uma probabilidade armazenada no atributo `pmut`. O código desse novo método é:

```
def selecao(self):
    nmut = array([i for i in range(self.npop) if random() < self
.pmut])
    return nmut
```

Usando o comando `nmut = array([i for i in range(n) if random() < pmut])` é realizada a escolha de quais indivíduos sofrerão mutação. Observe que, quanto maior for o valor de `pmut`, maior será o número de mutações ocorridas na população. Vale ressaltar que a função `random()` gera um número aleatório entre 0 e 1. Assim o valor de `pmut` precisa ser menor ou igual a um. O código completo da classe será:

```
#!/usr/bin/env python3.6
# -*- Coding: UTF-8 -*-
"""
Mutação.

Programa sob licença GNU V.3.
Desenvolvido por: E. S. Pereira.
Versão 0.0.1.
"""

from numpy.random import randint, random
from numpy import array
```

```
class Mutacao:
    """
    Classe base para operadores de mutação:
    Entrada:
        pmut - probabilidade de ocorrer uma mutação.
    """

    def __init__(self, pmut):
        self.pmut = pmut
        self._populacao = None
        self.npop = None
        self.ngen = None

    def _set_populacao(self, populacao):
        self._populacao = populacao
        self.npop = self._populacao.shape[0]
        self.ngen = self._populacao.shape[1]

    def _get_populacao(self):
        return self._populacao

    def selecao(self):
        nmut = array([i for i in range(self.npop) if random() <
self.pmut])
        return nmut

    def mutacao(self):
        raise NotImplementedError("A ser implementado")

    populacao = property(_get_populacao, _set_populacao)
```

Observe que o atributo `_populacao` é inicializado como `None`. Isso indica que, logo após instanciarmos um objeto do tipo `Mutacao`, seremos obrigados a inicializar o atributo `populacao`. Por exemplo:

```
mutacao = Mutacao(pmut=0.5)
mutacao.populacao = populacao
```

Na próxima seção veremos como estender a classe `Mutacao` para gerar o operador *flip*.

5.1 MUTAÇÃO FLIP

É uma mutação que transforma aleatoriamente o bit de um gene em outro. Por exemplo, se o bit escolhido for 0, ele mudará para 1 e vice-versa.

Considere o seguinte indivíduo, 1010010110100110. Agora vamos dizer que o quarto bit foi escolhido para sofrer mutação, o novo indivíduo será: 1011010110100110.

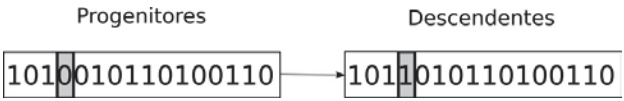


Figura 5.1: Mutação Flip no quarto bit.

Vamos criar o arquivo `flip.py` e importar os módulos e funções necessários:

```
from numpy.random import randint
from numpy import array

from .mutacao import Mutacao
```

A nova classe herdará os atributos e métodos da classe `Mutacao`.

```
class Flip(Mutacao):
    """
    Mutaçao flip.

    Entrada:
        pmut - probabilidade de ocorrer uma mutação.
    """
    def __init__(self, pmut):
        super(Mutacaoflip, self).__init__(pmut)
```

Aqui vamos implementar efetivamente o método `mutacao`:

```
def mutacao(self):
    """Alteração genética de membros da população usando mutação
flip."""
    nmut = self.selecao()
    genflip = array([randint(0, self.ngen - 1) for _ in nmut])
    self.populacao[nmut, genflip] = 1-self.populacao[nmut, genfli
p]
```

A variável `nmut` contém o índice dos indivíduos escolhidos para sofrer mutação. Logo em seguida, na variável `genflip`, são escolhidos quais genes serão modificados. Ao se fazer a operação `1-populacao[nmut, genflip]` estamos invertendo os zeros por uns e vice-versa (`1-0=1` e `1-1=0`).

O código completo contido no arquivo `flip.py` será:

```
#!/usr/bin/env python3.6
# -*- Coding: UTF-8 -*-
"""
Mutação.

Programa sob licença GNU V.3.
Desenvolvido por: E. S. Pereira.
Versão 0.0.1.
"""

from numpy.random import randint
from numpy import array

from .mutacao import Mutacao


class Flip(Mutacao):
    """
    Mutaçao flip.

    Entrada:
        pmut - probabilidade de ocorrer uma mutação.
    """
    def __init__(self, pmut):
        super(Mutacaoflip, self).__init__(pmut)
```

```
    def mutacao(self):
        """Alteração genética de membros da população usando muta
ção flip."""
        nmut = self.selecao()
        genflip = array([randint(0, self.ngen - 1) for _ in nmut]
)
        self.populacao[nmut, genflip] = 1-self.populacao[nmut, ge
nflip]
```

5.2 MUTAÇÃO DE DUPLA TROCA

A mutação de dupla troca realiza a troca de posição de dois genes escolhidos aleatoriamente. Por exemplo, considere a seguinte cadeia genética: 1010010110100110. Vamos supor que foram escolhidos o quarto e o décimo primeiro gene para troca. O descendente nesse caso será: 1011010110000110.

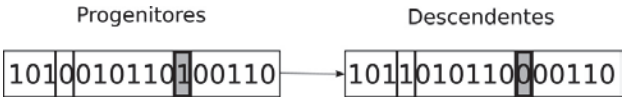


Figura 5.2: Mutação de dupla troca. Mudança de posição do quarto bit com o décimo primeiro bit. bit.

No módulo `mutacao` vamos criar o arquivo `duplatroca.py`, contendo a classe `DuplaTroca`. Vamos realizar as seguintes importações:

```
from numpy.random import randint
from numpy import array

from .mutacao import Mutacao
```

Tal como para a classe `Flip`, nossa nova classe será:

```
class DuplaTroca(Mutacao):
    """
    Mutaçao dupla troca.
```

```
    Entrada:
        populacao - vetor de população que deverá sofrer mutação.
        pmut - probabilidade de ocorrer uma mutação.
    """
    def __init__(self, pmut):
        super(DuplaTroca, self).__init__(pmut)
```

A seguir temos a implementação do método `mutacao`:

```
def mutacao(self):
    """Alteração genética de membros da população usando dupla tr
oca."""
    nmut = self.selecao()

    gen1 = array([randint(0, self.ngen - 1)])
    gen2 = array([randint(0, self.ngen - 1)])

    self.populacao[nmut, gen1], self.populacao[nmut, gen2] = \
    self.populacao[nmut, gen2], self.populacao[nmut, gen1]
```

Primeiramente, escolhemos quais indivíduos da população sofreram mutação. O próximo passo é escolher os genes que serão usados para troca, armazenados nas variáveis `gen1` e `gen2`. A última linha realiza a troca (*swap*). Observe que a `\` é usada para indicar que a próxima linha é continuação da linha anterior. Assim, membros da população com os genes `gen1` passaram a ter os genes `gen2` e vice-versa.

O código completo contido no arquivo `duplatroca.py` será:

```
#!/usr/bin/env python3.6
# -*- Coding: UTF-8 -*-
"""
Mutação dupla troca.

Programa sob licença GNU V.3.
Desenvolvido por: E. S. Pereira.
Versão 0.0.1.
"""

from numpy.random import randint
```

```
from numpy import array

from .mutacao import Mutacao


class DuplaTroca(Mutacao):
    """
    Mutação dupla troca.

    Entrada:
        pmut - probabilidade de ocorrer uma mutação.
    """
    def __init__(self, populacao, pmut):
        super(DuplaTroca, self).__init__(pmut)

    def mutacao(self):
        """Alteração genética de membros da população usando dup
la troca."""
        nmut = self.selecao()

        gen1 = array([randint(0, self.ngen - 1) for _ in nmut])
        gen2 = array([randint(0, self.ngen - 1) for _ in nmut])

        self.populacao[nmut, gen1], self.populacao[nmut, gen2] =
 \
        self.populacao[nmut, gen2], self.populacao[nmut, gen1]
```

5.3 MUTAÇÃO DE SEQUÊNCIA REVERSA

Na mutação reversa, uma sequência genética de um indivíduo, dentro de um dado intervalo, sofre uma inversão. Por exemplo, considere o seguinte cromossomo: 1000011111100110. Agora, assuma que o intervalo entre o bit 3 ao 8 foi escolhido. Nesse caso, o novo indivíduo será: 10111000100110.

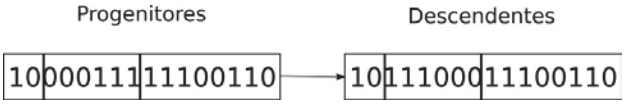


Figura 5.3: Mutação de sequência reversa. O intervalo indo do bit 3 ao bit 8 é escolhido para sofrer reversão. bit.

Dentro do pacote `mutacao` vamos criar o arquivo `sequenciareversa.py`. Nesse módulo, as importações, juntamente com a declaração inicial da classe `SequenciaReversa`, serão:

```
from numpy.random import randint
from numpy import array

from .mutacao import Mutacao


class SequenciaReversa(Mutacao):
    """
    Mutaçao Sequência Reversa.

    Entrada:
        pmut - probabilidade de ocorrer uma mutação.
    """
    def __init__(self, pmut):
        super(SequenciaReversa, self).__init__(pmut)
```

Que é similar aos casos anteriores. Já a implementação do método `mutacao` é dada por:

```
def mutacao(self):
      """
      Alteração genética de membros da população usando sequência
 reversa.
      """
      nmut = self.selecao()
      if nmut.size != 0:
          for k in nmut:
              i = randint(0, self.ngen - 1)
              j = randint(0, self.ngen - 1)
              while i == j:
                  j = randint(0, self.ngen - 1)
              if i > j:
                  i, j = j, i
              self.populacao[k, i:j] = self.populacao[k, i:j][::-
]
```

Primeiramente, o método armazena os índices dos indivíduos

que vão sofrer mutação. Como os intervalos de mutação serão distintos para cada indivíduo, temos que fazer a operação de reversão de forma individual. Para isso, criou-se um laço de repetição `for k in mut:` .

O próximo passo foi o de se determinar o intervalo de interesse. Como a escolha é aleatória, pode ocorrer que o valor de `i` e `j` sejam iguais. Nesse caso, um novo laço de repetição foi utilizado para garantir que esse fato não ocorra. Como `i` é o ponto inicial, verificamos se `i` é maior que `j` . Caso isso ocorra, uma troca é realizada e `i` se torna `j` e vice-versa. Ao se utilizar o termo `[::-1]` estamos obtendo uma reversão de um vetor. Dessa forma, conseguimos cumprir o objetivo do operador.

A seguir é apresentado o código completo do módulo `sequenciareversa.py`

```
#!/usr/bin/env python3.6
# -*- Coding: UTF-8 -*-
"""
Mutação flip.

Programa sob licença GNU V.3.
Desenvolvido por: E. S. Pereira.
Versão 0.0.1.
"""

from numpy.random import randint
from numpy import array, newaxis

from .mutacao import Mutacao


class SequenciaReversa(Mutacao):
    """
    Mutaçao Sequência Reversa.

    Entrada:
```

```
        pmut - probabilidade de ocorrer uma mutação.
    """
    def __init__(self, pmut):
        super(SequenciaReversa, self).__init__(pmut)

    def mutacao(self):
        """
        Alteração genética de membros da população usando sequênc
ia reversa.
        """
        nmut = self.selecao()
        if nmut.size != 0:
            for k in nmut:
                i = randint(0, self.ngen - 1)
                j = randint(0, self.ngen - 1)
                while i == j:
                    j = randint(0, self.ngen - 1)
                if i > j:
                    i, j = j, i
                self.populacao[k, i:j] = self.populacao[k, i:j][:
:-1]
```

Conclusão

Neste capítulo foram apresentados os operadores genéticos de mutação *flip*, por dupla troca e de sequência reversa. Nesse caso, as operações de mutação ocorrem na própria população. Dessa forma, usamos a passagem por referência para realizar as modificações necessárias. Também foi apresentado que o objetivo principal da mutação é manter a variabilidade genética da população, permitindo que a busca não fique presa a uma solução local.

No próximo capítulo será mostrado como combinar os operadores genéticos para fazer a população evoluir, buscando a melhor solução possível para o problema em que se está interessado.

Capítulo 6

EVOLUÇÃO

Nos capítulos anteriores foram apresentados os operadores de seleção (para a escolha dos indivíduos que vão propagar seus genes), cruzamento (usado para combinar soluções e gerar descendentes) e mutação (que faz pequenas alterações nas soluções com a finalidade de ampliar o espaço de busca, evitando que a população fique presa em soluções locais).

Neste capítulo veremos como combinar esses operadores para evoluir as soluções iniciais e chegar a uma solução ótima para um dado problema.

A estrutura atual do projeto `pygenec` será:

```
pygenec\
 |
 |-pygenec\
           |-__init__.py
           |-populacao.py
           |-evolucao.py
           |-selecao\
                    |-__init__.py
                    |-roleta.py
                    |-classificacao.py
                    |-torneio.py
           |-cruzamento\
                    |-__init__.py
                    |-cruzamento.py
                    |-embaralhamento.py
```

```
                |-kpontos.py
                |-umponto.py
        |-mutacao\
                |-__init__.py
                |-mutacao.py
                |-duplatroca.py
                |-flip.py
                |-sequenciareversa.py
|-main.py
```

6.1 A SOBREVIVÊNCIA DO MAIS ADAPTADO: INTEGRANDO OS OPERADORES GENÉTICOS

Os operadores de cruzamento utilizados são considerados elitistas, pois privilegiam a propagação genética das soluções com maior valor *fitness*. Outra estratégia elitista que usaremos aqui será a de manter intactos os genes do melhor indivíduo. Isso significa que, a cada geração, antes de aplicar os operadores genéticos, vamos verificar qual solução tem o maior valor *fitness* e o indivíduo que representa essa solução será mantido sem alteração na próxima geração. Isso não significa que ele se manterá intacto do início ao fim da evolução, apenas garantimos que ao menos uma boa solução sempre será mantida ao se passar de uma geração para a seguinte.

No pacote `pygenec`, vamos criar o arquivo `evolucao.py` contendo a classe `Evolucao`, cuja declaração e seu construtor será:

```
class Evolucao:
    """
    Usando operadores genéticos, coloca uma população para evolui
r.

    Entrada:
        populacao - Objeto do tipo Populacao
```

```
        selecao - Objeto do tipo Selecao
        cruzamento - Objeto do tipo cruzamento
        mutacao - Objeto do tipo  Mutacao.
    """

    def __init__(self, populacao, selecao, cruzamento, mutacao):
        self.populacao = populacao
        self.selecao = selecao
        self.cruzamento = cruzamento
        self.mutacao = mutacao

        self._geracao = 0
        self._melhor_solucao = None
        self._nsele = None
        self._pcruz = None
```

Essa classe recebe no construtor objetos do tipo `Populacao`, `Selecao`, `Cruzamento` e `Mutacao`. Observe que aqui vamos utilizar o polimorfismo da Programação Orientada a Objetos. Por exemplo, aqui o operador descrito pela classe `KPontos` será igualmente tratado com respeito às classes `UmPonto` ou `Embaralhamento`. Os atributos privados `_geracao`, `_melhor_solucao`, `_nsele` e `_pcruz` representam, respectivamente, a geração atual, o indivíduo com maior valor *fitness* da rodada, o número de indivíduos a ser selecionados para o cruzamento e a probabilidade de cruzamento.

O próximo passo é definir os métodos *getters* e *setters* dos atributos privados:

```
def _set_nsele(self, nsele):
    self._nsele = nsele

def _get_nsele(self):
    return self._nsele

def _set_pcruz(self, pcruz):
    self._pcruz = pcruz
```

```
def _get_pcruz(self):
    return self._pcruz

@property
def melhor_solucao(self):
    return self._melhor_solucao

@property
def geracao(self):
    return self._geracao
```

Usando somente o decorador `@property`, estamos indicando que os atributos `_melhor_solucao` e `_geracao` não podem ser alterados externamente à classe, sendo acessados apenas como leitura. Ao final da classe, adicionamos o acesso aos outros atributos:

```
nsele = property(_get_nsele, _set_nsele)
pcruz = property(_get_pcruz, _set_pcruz)
```

O método `evoluir` será:

```
def evoluir(self):
    """
    Evolução elitista, por uma geração, da população.
    """
    if self._first is True:
        self._fitness = self.populacao.avaliar()
        self._first = False

    self._melhor_solucao = self.populacao.populacao[-1].copy()

    subpopulacao = self.selecao.selecao(self._nsele, fitness=self
._fitness)
    populacao = self.cruzamento.descendentes(subpopulacao, pcruz=
self._pcruz)

    self.mutacao.populacao = populacao
    self.mutacao.mutacao()
    self.populacao.populacao[:] = populacao[:]

    self._geracao += 1
```

```
    if self._manter_melhor is True:
        self.populacao.populacao[0] = self._melhor_solucao

    self._fitness = self.populacao.avaliar()

    return self._fitness.min(), self._fitness.max()
```

Após a declaração do método, na primeira linha, é realizada uma avaliação da população corrente. A melhor solução é então armazenada no atributo `_melhor_solucao`. Usando o operador de seleção, uma subpopulação é criada. A partir do cruzamento, passamos a ter uma nova população. Esses novos indivíduos passam então pelo processo de mutação. Em seguida, o atributo `populacao` é atualizado, sendo que o primeiro indivíduo será representado pela melhor solução da população original. Para finalizar, o atributo `_geracao` é atualizado.

O código completo contido no módulo `evolucao.py` é apresentado a seguir:

```
#!/usr/bin/env python3.6
# -*- Coding: UTF-8 -*-
"""
Evolução.

Programa sob licença GNU V.3.
Desenvolvido por: E. S. Pereira.
Versão 0.0.1.
"""


class Evolucao:
    """
    Usando operadores genéticos, coloca uma população para evolui
r.

    Entrada:
        populacao - Objeto do tipo Populacao
        selecao - Objeto do tipo Selecao
```

```
        cruzamento - Objeto do tipo Cruzamento
        mutacao - Objeto do tipo Mutacao.
    """

    def __init__(self, populacao, selecao, cruzamento, mutacao):
      self.populacao = populacao
      self.selecao = selecao
      self.cruzamento = cruzamento
      self.mutacao = mutacao
      self._melhor_solucao = None
      self._geracao = 0
      self._nsele = None
      self._pcruz = None
      self._manter_melhor = True
      self._fitness = None
      self._first = True

    def _set_nsele(self, nsele):
        self._nsele = nsele

    def _get_nsele(self):
        return self._nsele

    def _set_pcruz(self, pcruz):
        self._pcruz = pcruz

    def _get_pcruz(self):
        return self._pcruz

    def _get_first(self):
          return self._first

      def _set_first(self, first):
          self._first = first

    @property
    def melhor_solucao(self):
        return self._melhor_solucao

    @property
    def geracao(self):
        return self._geracao

    def evoluir(self):
        """
```

```
        Evolução elitista, por uma geração, da população.
        """
        if self._first is True:
            self._fitness = self.populacao.avaliar()
            self._first = False

        self._melhor_solucao = self.populacao.populacao[-1].copy(
)

        subpopulacao = self.selecao.selecao(self._nsele, fitness=
self._fitness)
        populacao = self.cruzamento.descendentes(subpopulacao, pc
ruz=self._pcruz)

        self.mutacao.populacao = populacao
        self.mutacao.mutacao()
        self.populacao.populacao[:] = populacao[:]

        self._geracao += 1

        if self._manter_melhor is True:
            self.populacao.populacao[0] = self._melhor_solucao

        self._fitness = self.populacao.avaliar()

        return self._fitness.min(), self._fitness.max()

    nsele = property(_get_nsele, _set_nsele)
    pcruz = property(_get_pcruz, _set_pcruz)
    first = property(_get_first, _set_first)
```

6.2 ENCONTRANDO O MÁXIMO

Vamos modificar o programa `main.py` para encontrar o máximo da função objetivo, que já foi apresentada no capítulo 2 e é representada pela seguinte figura:

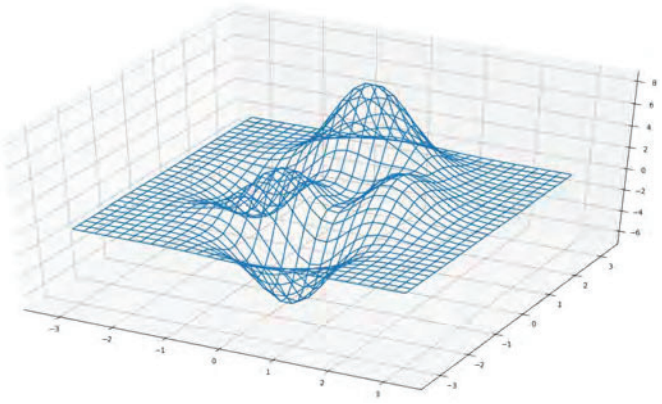


Figura 6.1: Função objetivo

Vamos importar os seguintes módulos:

```
from numpy import exp, array, mgrid
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
from matplotlib.animation import FuncAnimation

from pygenec.populacao import Populacao
from pygenec.selecao.roleta import Roleta
from pygenec.cruzamento.kpontos import KPontos
from pygenec.mutacao.flip import Flip
from pygenec.evolucao import Evolucao
```

Aqui será utilizada a função `FuncAnimation`, que permite gerar gráficos animados para exibir o processo de evolução na busca do máximo da função. Vamos manter as mesmas funções já utilizadas anteriormente, assim, a nova modificação terá os seguintes objetos:

```
cromossos_totais = 32
tamanho_populacao = 50

populacao = Populacao(avaliacao, cromossos_totais, tamanho_popula
cao)
```

```
selecao = Roleta(populacao)
cruzamento = KPontos(tamanho_populacao)
mutacao = Flip(pmut=0.9)
evolucao = Evolucao(populacao, selecao, cruzamento, mutacao)

evolucao.nsele = 10
evolucao.pcruz = 0.5
```

Note que agora adicionamos o objeto `evolucao`, definindo em seguida os atributos `nsel`, com o valor 10, e `pcruz`, como `0.5`.

O próximo código representa a inicialização do gráfico, o qual já foi apresentado no capítulo 2:

```
fig = plt.figure(figsize=(100, 100))
ax = fig.add_subplot(111, projection="3d")
X, Y = mgrid[-3:3:30j, -3:3:30j]
Z = func(X,Y)
ax.plot_wireframe(X, Y, Z)
```

O gráfico que representa as soluções encontradas via algoritmo genético é:

```
x, y = xy(populacao.populacao)
z = func(x, y)
graph = ax.scatter(x, y, z, s=50, c='red', marker='D')
```

Note que criamos a variável `graph`, o qual armazena o gráfico de pontos espalhados (*scatter*). Esses pontos vão representar os indivíduos da população, lembrando que a posição do ponto sobre o gráfico de arame representa a busca pelo máximo da função. Os indivíduos mais adaptados serão aqueles que estiverem nos pontos mais elevados da figura que representa a função a ser explorada.

Para gerar a animação, precisamos criar a função `update`. Dentro dessa função será executado o método `evoluir` do objeto `evolucao`. Logo, rodando esse método `n` vezes, a evolução

ocorrerá por `n` gerações:

```
def update(frame):
    evolucao.evoluir()
    x, y = xy(populacao.populacao)
    z = func(x, y)
    graph._offsets3d = (x, y, z)
```

Note que a cada *frame* rodamos o algoritmo genético e comparamos a população com relação à função objetivo (`evolucao.evoluir()`). Utilizando o comando `graph._offsets3d = (x, y, z)` atualizamos os dados do gráfico de pontos.

Para finalizar, chamamos a função de animação:

```
ani = FuncAnimation(fig, update, frames=range(10000), repeat=Fals
)
plt.show()
```

Essa função recebe a figura a ser atualizada, a função `update` e uma lista contendo o total de *frames* da animação. Além disso, definimos `repeat=False` para indicar que não queremos reiniciar a animação automaticamente. Isso é feito pois `frames=range(10000)` representa que queremos rodar o código por dez mil gerações. Se repetirmos a animação, a evolução vai continuar sem parar, finalizando apenas quando fecharmos a janela do gráfico. Para saber mais sobre a função `FuncAnimation`, veja a documentação oficial em https://matplotlib.org/api/_as_gen/matplotlib.animation.FuncAnimation.html/.

O código completo do arquivo `main.py` é:

```
#!/usr/bin/env python3.6
# -*- Coding: UTF-8 -*-
"""
```

```
Obtenção de máximo de função.

Programa sob licença GNU V.3.
Desenvolvido por: E. S. Pereira.
Versão 0.0.1.
"""
from numpy import exp, array, mgrid
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
from matplotlib.animation import FuncAnimation

from pygenec.populacao import Populacao
from pygenec.selecao.roleta import Roleta
from pygenec.cruzamento.kpontos import KPontos
from pygenec.mutacao.flip import Flip
from pygenec.evolucao import Evolucao



def func(x, y):
    tmp = 3 * exp(-(y + 1) ** 2 - x **2)*(x - 1)**2 \
          - (exp(-(x+ 1) ** 2 - y **2) / 3 )\
          + exp(-x **2 - y ** 2) * (10 * x **3 - 2 * x + 10 * y *
* 5)
    return tmp


def bin(x):
    cnt = array([2 ** i for i in range(x.shape[1])])
    return array([(cnt * x[i,:]).sum() for i in range(x.shape[0])
])


def xy(populacao):
    colunas = populacao.shape[1]
    meio = int(colunas / 2)
    const = 2.0 ** meio - 1.0
    nmin = -3
    nmax = 3
    const = (nmax - nmin) / const
    x = nmin + const * bin(populacao[:,:meio])
    y = nmin + const * bin(populacao[:,meio:])
    return x, y
```

```
def avaliacao(populacao):
    x, y = xy(populacao)
    tmp = func(x, y)
    return tmp


cromossos_totais = 32
tamanho_populacao = 50

populacao = Populacao(avaliacao, cromossos_totais, tamanho_popula
cao)
selecao = Roleta(populacao)
cruzamento = KPontos(tamanho_populacao)
mutacao = Flip(pmut=0.9)
evolucao = Evolucao(populacao, selecao, cruzamento, mutacao)

evolucao.nsele = 10
evolucao.pcruz = 0.5


fig = plt.figure(figsize=(100, 100))
ax = fig.add_subplot(111, projection="3d")
X, Y = mgrid[-3:3:30j, -3:3:30j]
Z = func(X,Y)
ax.plot_wireframe(X, Y, Z)

x, y = xy(populacao.populacao)
z = func(x, y)
graph = ax.scatter(x, y, z, s=50, c='red', marker='D')

def update(frame):
    evolucao.evoluir()
    x, y = xy(populacao.populacao)
    z = func(x, y)
    graph._offsets3d = (x, y, z)

ani = FuncAnimation(fig, update, frames=range(10000), repeat=Fals
)
plt.show()
```

Nas próximas figuras são apresentadas as soluções obtidas com o código anterior. A primeira figura representa a execução inicial do programa, já as duas figuras a seguir foram produzidas após dez

gerações. Nesse caso, podemos observar que a população rapidamente converge para o máximo da função, encontrando a solução que esperávamos para o problema.

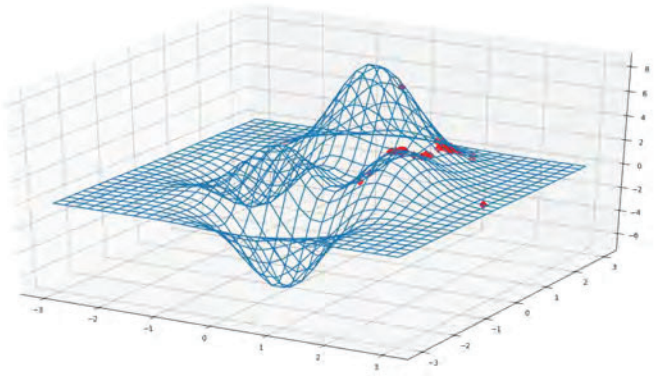


Figura 6.2: Estado inicial das possíveis soluções representadas pela população.

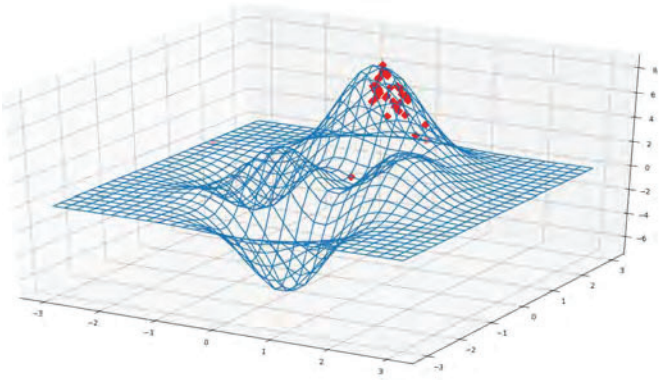


Figura 6.3: Soluções após dez gerações.

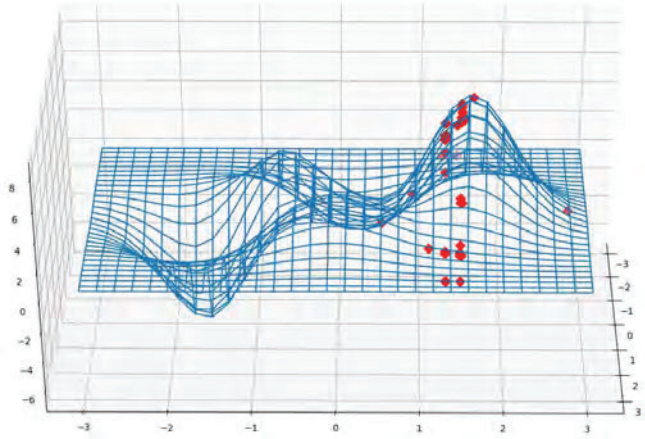


Figura 6.4: Vista lateral, mostrando que já temos soluções no máximo da função após dez gerações.

Na próxima figura é apresentado um caso em que a solução ficou presa em um máximo local. Esse caso foi obtido ao não se comentar o código que mantém indivíduo com maior *fitness* na geração seguinte (comentando o comando `self.populacao.populacao[0] = self._melhor_solucao`).

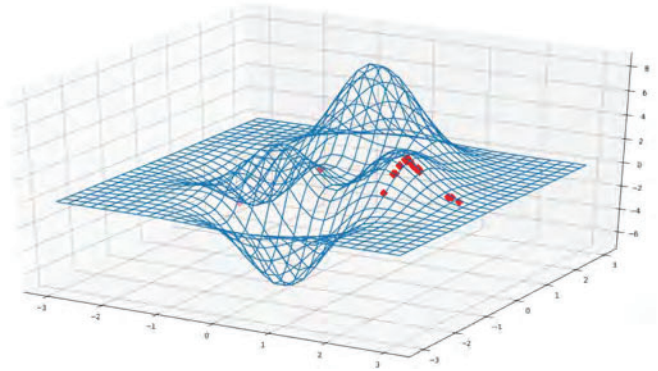


Figura 6.5: População presa em um máximo local.

O próximo resultado foi obtido mudando a função de avaliação para buscar o mínimo da função. Para encontrar o mínimo basta multiplicar a função objetivo por "menos um" (`-1`):

```
def avaliacao(populacao):
    x, y = xy(populacao)
    tmp = -func(x, y)
    return tmp
```

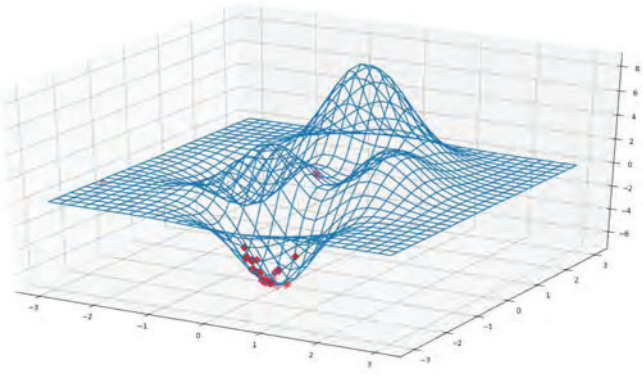


Figura 6.6: Determinação do mínimo da função.

Conclusão

Neste capítulo, combinamos os operadores genéticos de seleção, mutação e cruzamento para efetivamente simular o processo de evolução, tendo como critério de adaptabilidade a função objetivo, cujo ponto máximo ou mínimo se deseja estimar. Vimos também que o uso de uma estratégia elitista evita que a população fique presa numa solução local.

No próximo capítulo vamos apresentar um outro operador ligado ao processo de evolução, conhecido como operador epidêmico.

Capítulo 7

A EVOLUÇÃO ATRAVÉS DA TRAGÉDIA

Ao longo da história de vida do planeta, por diversas vezes ocorreram dizimações de várias espécies. Algumas acabaram por ser extintas, permitindo que outras pudessem proliferar. Mesmo em períodos recentes da humanidade, caso de epidemias levaram a vida de muitas pessoas. Quando uma doença séria ataca uma espécie, mesmo indivíduos bem adaptados a ao ambiente podem não sobreviver, o que conduz a uma mudança drástica da população. Os sobreviventes acabam desenvolvendo imunidade e passando essa resistência a gerações posteriores. Caso haja sobreviventes, esse processo pode acelerar a evolução de uma espécie.

No caso de problemas de optimização, envolvendo algoritmos genéticos, a população pode ficar "presa" em um mínimo/máximo local, não sendo mais capaz de encontrar a melhor solução possível para um dado problema. Em outros casos, para acelerar a busca é preciso testar um número maior de soluções possíveis do que o representado pela população atual. Em ambas as situações, pode ser importante reiniciar parte da população depois de um certo número de passos evolutivos. Inspirados na possibilidade de evolução rápida devido à sujeição de espécies a epidemias,

Chiwiacowsky e Velho (2003) chamaram essa reinicialização de operador epidêmico.

Neste capítulo vamos modificar a classe `Evolucao` para adicionar esse operador.

7.1 OPERADOR EPIDÊMICO

A ideia básica desse operador é que, após um determinado número de passos evolutivos, a maior parte da população seja dizimada, sobrevivendo apenas uma pequena porcentagem de indivíduo com melhor valor *fitness*. Na prática, o que se faz é criar uma população e em seguida substituir parte de seus indivíduos por alguns dos melhores da população que sofrerá morte por epidemia. Com isso, ampliamos o espaço de busca, mantendo vivas as melhores soluções encontradas até o momento. A escolha de quando esse operador deve ser executado será determinada via experimento computacional, podendo variar de caso para caso. Também vamos parametrizar a decisão entre manter ou não o melhor indivíduo na próxima geração.

Vamos modificar a classe evolução para adicionar esse novo operador:

```
class Evolucao:
    """
    Usando operadores genéticos, coloca uma população para evolui
r.

    Entrada:
        populacao - Objeto do tipo Populacao
        selecao - Objeto do tipo Selecao
        cruzamento - Objeto do tipo Cruzamento
        mutacao - Objeto do tipo Mutacao.
    """
```

```
    def __init__(self, populacao, selecao, cruzamento, mutacao):
        self.populacao = populacao
        self.selecao = selecao
        self.cruzamento = cruzamento
        self.mutacao = mutacao
        self._melhor_solucao = None
        self._geracao = 0
        self._nsele = None
        self._pcruz = None
        self._epidemia = None
        self._manter_melhor = True
        self._first = True

    def _set_epidemia(self, epidemia):
        self._epidemia = int(epidemia)

    def _set_manter_melhor(self, manter):
        self._manter_melhor = manter

    def _get_manter_melhor(self):
        return self._manter_melhor

    def _get_epidemia(self):
        return self._epidemia

    def _get_first(self):
          return self._first

      def _set_first(self, first):
          self._first = first
```

Ao final da classe, vamos definir os *properties*:

```
epidemia = property(_get_epidemia, _set_epidemia)
manter_melhor = property(_get_manter_melhor, _set_manter_melhor)
```

Dentro do método `evoluir` adicionaremos o código de epidemia:

```
if self._epidemia is not None:
    if self._geracao % self._epidemia == 0 and random() < 0.8:
        """Passo Epidêmico"""
        print("Epidemia")
```

```
        self._possivel_local = 0
        self.populacao.gerar_populacao()
        self.populacao.populacao[0] = self._melhor_solucao
```

No código anterior vemos que, se o atributo `_epidemia` não for do tipo `None`, a cada vez que o número de gerações executadas for um múltiplo dessa variável (`self._epidemia`), uma nova população será gerada. Note que deixamos a probabilidade de ocorrência de epidemia em 80% (`random() < 0.8`) em vez de sempre ocorrer. Como nosso método é estocástico, é interessante deixar uma margem para que a epidemia não ocorra. Ao ativar o operador epidêmico, a probabilidade deve ser elevada, pois queremos garantir a ampliação do espaço de busca da solução. O leitor pode fazer testes com outros valores e avaliar se isso acelera ou não o processo de busca da solução. Na última linha do código anterior, fazemos com que o primeiro membro da nova população seja a melhor solução da geração anterior, preservando o indivíduo mais adaptado até o momento.

O código completo da classe será:

```
#!/usr/bin/env python3.6
# -*- Coding: UTF-8 -*-
"""
Evolução.

Programa sob licença GNU V.3.
Desenvolvido por: E. S. Pereira.
Versão 0.0.1.
"""

from numpy.random import random


class Evolucao:
    """
    Usando operadores genéticos, coloca uma população para evolui
r.
```

```
    Entrada:
        populacao - Objeto do tipo Populacao
        selecao - Objeto do tipo Selecao
        cruzamento - Objeto do tipo Cruzamento
        mutacao - Objeto do tipo Mutacao.
    """

    def __init__(self, populacao, selecao, cruzamento, mutacao):
        self.populacao = populacao
        self.selecao = selecao
        self.cruzamento = cruzamento
        self.mutacao = mutacao
        self._melhor_solucao = None
        self._geracao = 0
        self._nsele = None
        self._pcruz = None
        self._epidemia = None
        self._manter_melhor = True
        self._first = True

    def _set_epidemia(self, epidemia):
        self._epidemia = int(epidemia)

    def _set_manter_melhor(self, manter):
        self._manter_melhor = manter

    def _get_manter_melhor(self):
        return self._manter_melhor

    def _get_epidemia(self):
        return self._epidemia

    def _set_nsele(self, nsele):
        self._nsele = nsele

    def _get_nsele(self):
        return self._nsele

    def _set_pcruz(self, pcruz):
        self._pcruz = pcruz

    def _get_pcruz(self):
        return self._pcruz
```

```
    def _get_first(self):
          return self._first

    def _set_first(self, first):
        self._first = first

    @property
    def melhor_solucao(self):
        return self._melhor_solucao

    @property
    def geracao(self):
        return self._geracao

    def evoluir(self):
        """
        Evolução elitista, por uma geração, da população.
        """
        fitness = self.populacao.avaliar()
        self._melhor_solucao = self.populacao.populacao[-1].copy(
)

        subpopulacao = self.selecao.selecao(self._nsele, fitness)
        populacao = self.cruzamento.descendentes(subpopulacao, pc
ruz=self._pcruz)

        self.mutacao.populacao = populacao
        self.mutacao.mutacao()
        self.populacao.populacao[:] = populacao[:]

        if self._manter_melhor is True:
            self.populacao.populacao[0] = self._melhor_solucao

        self._geracao += 1

        if self._epidemia is not None:
            if self._geracao % self._epidemia == 0 and random() <
0.8:
                """Passo Epidêmico"""
                print("Epidemia")
                self._possivel_local = 0
                self.populacao.gerar_populacao()
                self.populacao.populacao[0] = self._melhor_soluca
o
```

```
        fitness = self.populacao.avaliar()
        return fitness.min(), fitness.max()

    nsele = property(_get_nsele, _set_nsele)
    pcruz = property(_get_pcruz, _set_pcruz)
    epidemia = property(_get_epidemia, _set_epidemia)
    manter_melhor = property(_get_manter_melhor, _set_manter_melh
or)
    manter_melhor = property(_get_manter_melhor, _set_manter_melh
or)
    first = property(_get_first, _set_first)
```

Conclusão

Neste capítulo foi apresentado o operador epidêmico, o qual é utilizado para ajudar na ampliação do espaço de busca por possíveis soluções. Esse operador foi adicionado à classe `Evolucao`, responsável pela execução do algoritmo genético.

Capítulo 8

DISPONIBILIZAÇÃO DO PACOTE

O objetivo principal deste capítulo é apresentar uma maneira disponibilizar os pacotes criados para a comunidade via comando `pip install` . O módulo também estará disponível em https://github.com/duducosmos/pygenec/.

Para tornar o pacote acessível via `pip` , o primeiro passo será criar uma conta no site *The Python Package Index* (PyPi), em https://pypi.org/. Em seguida, vamos organizar a estrutura do módulo e adicionar os elementos necessários para instalação do pacote na máquina dos usuários.

8.1 ORGANIZAÇÃO DO PACOTE

Neste passo, vamos criar pasta `pygenec_pypi` , com a seguinte estrutura:

```
pygenec_pypi\
             |-LICENSE
             |-README.md
             |-setup.py
             |-src\
                   |-pygenec\
                             |-__init__.py
                             |-populacao.py
```

```
|-evolucao.py
|-tools.py
|-selecao\
         |-__init__.py
         |-roleta.py
         |-classificacao.py
         |-torneio.py
         |-selecao.py
|-cruzamento\
         |-__init__.py
         |-cruzamento.py
         |-embaralhamento.py
         |-kpontos.py
         |-umponto.py
 |-mutacao\
         |-__init__.py
         |-mutacao.py
         |-duplatroca.py
         |-flip.py
         |-ntrocas.py
         |-sequenciareversa.py
```

No arquivo `LICENSE` estará disponível a licença do pacote, que nesse caso será *Apache License 2.0*. O arquivo `README.md` conterá informações gerais do pacote. O arquivo `setup.py` contém informações sobre o processo de instalação. Esse programa contém o seguinte código:

```
#!/usr/bin/env python3
# -*- Coding: UTF-8 -*-

import os
from setuptools import setup, find_packages


def read(filename):
    return open(os.path.join(os.path.dirname(__file__), filename)
).read()

setup(
    name="pygenec",
    license="Apache License 2.0",
```

```
    version='1.0.0',
    author='Eduardo S. Pereira',
    author_email='pereira.somoza@gmail.com',
    packages=find_packages("src"),
    package_dir={"":"src"},
    description="Algoritmo Genetico em Python e Numpy",
    long_description=read("README.md"),
    long_description_content_type="text/markdown",
    url="https://github.com/duducosmos/pygenec",
    include_package_data=True,
    install_requires=["numpy"]
)
```

Primeiramente, importamos o módulo `os` e as funções do `setup`, `find_packages` do módulo `setuptools`. A função `read` é usada para adicionar o caminho completo do arquivo *README*.

8.2 SUBINDO O PACOTE PARA O PYPI

Para facilitar o processo de upload do pacote no PyPi, vamos instalar a ferramenta `twine`, que é um utilitário para publicação de pacotes de forma segura.

```
pip install twine
```

Uma vez instalado o utilitário de upload, vamos criar os arquivos de distribuição via `pip`. Faremos isso porque os módulos no PyPi não são distribuídos no formato de código-fonte e sim em pacotes de distribuição. Para criar tal pacote, usaremos o seguinte comando:

```
python setup.py sdist bdist_wheel
```

Assim, o arquivo de distribuição será construído a partir das informações contidas no arquivo `setup.py`.

O próximo passo é verificar se a distribuição foi gerada corretamente, nesse caso, usaremos a ferramenta `twine` , a partir do comando:

```
twine check dist/*
```

Se a verificação finalizou sem nenhum erro, para fazer o upload ao PyPi, usa-se o comando:

```
twine upload dist/*
```

Ao digitar esse o comando serão pedidos o nome de usuário e a senha. Se tudo der certo, o pacote poderá ser instalado diretamente usando o comando:

```
pip install pygenec
```

Conclusão

Neste capítulo, foi apresentado como disponibilizar o framework de algoritmos genéticos no repositório PyPi. Com isso, a nossa ferramenta ficou disponível para instalação via comando `pip install pygenec` . Na segunda parte do livro será apresentado um caso de estudo, o qual permitirá a você adquirir uma experiência concreta na utilização de algoritmos genéticos.

Este capítulo finaliza a primeira parte do livro, em que foram apresentados os operadores genéticos e suas principais variações. Na segunda parte, serão apresentadas aplicações de algoritmo genético no contexto de aprendizagem de máquina (*Machine Learning*).

Estudo de caso

Capítulo 9

EXPLORANDO O USO DE ALGORITMOS GENÉTICOS

De acordo com Pacheco (1999), algoritmos genéticos (AGs) têm aplicações nas áreas de problemas: complexos de otimização; com diversos parâmetros ou características a serem combinadas para encontrar a melhor solução; com grande espaço de busca ou muitas restrições. Algumas das aplicações citadas pelo autor são: determinação de melhor rota de veículos; design de circuitos; classificação de clientes; alocação de espaço físico; fluxo de caixa inteligente.

Para ser capaz de resolver classes de problemas diferentes com um mesmo método usamos o conceito de redutibilidade. Sipser (2007) apresenta que, na redutibilidade por mapeamento, um problema A pode ser reduzido ao problema B usando uma função computável que converte instâncias de B, ou seja, procuramos maneiras de converter B em A.

Por exemplo, viajar de São Paulo a Lisboa pode ser reduzido a encontrar rotas aéreas de uma cidade a outra, o qual se reduz ao problema de comprar passagens mais baratas. Outro caso é que a determinação de rotas de veículos e fluxo de caixa inteligente podem ser reduzidos ao problema de encontrar caminhos em

grafos conectados.

Nesta parte do livro, como estudo de caso, exploraremos o uso de computação evolucionária para encontrar caminhos em labirintos. Nesse exemplo nos debruçaremos sobre o conceito de redução do problema do labirinto em um grafo conectado. Assim, a melhor solução será aquela que, ao se partir de um ponto, chega ao objetivo com o menor número de passos possíveis. Este caso de estudo nos permitirá ter uma visão prática e didática de como utilizar algoritmos genéticos em problemas complexos e que vão além de encontrar parâmetros em funções.

Capítulo 10

GERANDO LABIRINTOS

Atualmente, o uso de navegação através de mapas digitais faz parte da realidade de muitas pessoas. Usando aplicativos ou sites específicos, procuramos qual a melhor rota, ou caminho mais curto, partindo do ponto A ao ponto B. Em pouco tempo, o aplicativo exibe em um mapa o melhor caminho a ser seguido.

Outra área que está em crescimento envolve a navegação de robôs em ambientes contendo obstáculos. Nesse caso, a máquina precisa ser capaz de ler ou criar um mapa e definir rotas internas de forma dinâmica. Em geral, os algoritmos de determinação de rotas são convertidos ao problema de encontrar caminhos em um grafo.

10.1 GRAFOS

Solucionar labirintos será reduzido ao problema de encontrar caminhos em grafos. Em linhas gerais, um grafo é formado por uma rede de pontos que podem ou não estar conectados. Os pontos são chamados de nós ou vértices, enquanto a linha que realiza a conexão é chamada de arestas. Podemos pensar nos vértices como estações do metrô, enquanto as arestas são os trilhos. Na figura a seguir é apresentado um exemplo de representação de

uma parte das linhas do metrô da cidade de São Paulo na forma de um grafo (Souza, 2016).

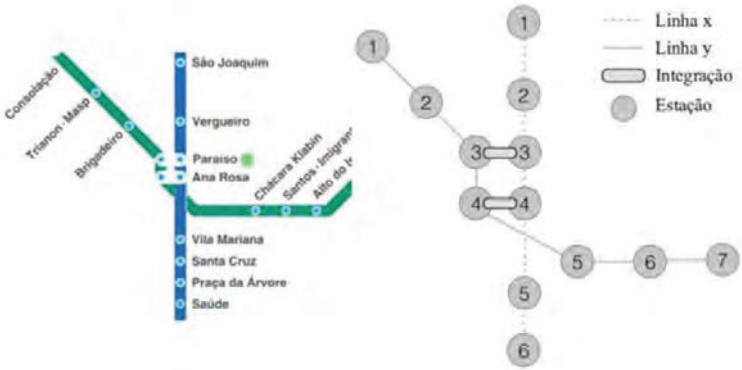


Figura 10.1: Representação de uma parte do metrô da cidade de São Paulo na forma de um grafo. Fonte: SOUZA, 2016

Na próxima figura é apresentado um grafo ponderado, em que cada aresta tem um peso. Tais pesos podem representar, por exemplo, a distância efetiva entre dois pontos.

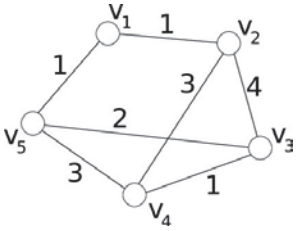


Figura 10.2: Grafo com pesos ponderados. Fonte: OLIVEIRA, 2013

Observe que o grafo da figura anterior é formado pelos conjuntos: vértices - $V = \{V_1, V_2, V_3, V_4, V_5\}$; arestas - $A = \{(V_1, V_2), (V_1, V_5), (V_2, V_3), (V_2, V_4), (V_3, V_4), (V_3, V_5), (V_4, V_5)\}$; pesos- $P = \{1, 1, 4, 3, 1, 2, 3\}$. O número de arestas em um vértice é

chamado de grau. Nesse exemplo, vemos que o vértice v_1 tem grau 2, enquanto o vértice V_5 tem grau 3.

Podemos representar o grafo ponderado usando uma matriz de adjacência em que linhas e colunas contêm o valor das arestas. Linha e coluna contendo valor infinito representam ausência de conexão:

	1	2	3	4	5
1	∞	1	∞	∞	1
2	1	∞	4	3	∞
3	∞	4	∞	1	2
4	∞	3	1	∞	3
5	1	∞	2	3	∞

De acordo com Sipzer (2007), uma sequência de vértices conectados por arestas é chamada de caminho, sendo que um caminho simples é aquele que não repete nenhum nó. Para resolver labirintos, faremos com que uma dada população represente caminhos simples que têm como primeiro vértice o ponto de partida. O melhor indivíduo será aquele que represente o caminho simples com o menor número de vértices conectando o local de partida ao ponto de interesse dentro do labirinto.

Os grafos apresentados anteriormente são chamados de grafos não direcionados. Existem também os chamados grafos direcionados, em que se tem uma direção específica para ir de um vértice a outro. Nesse caso, as arestas possuem setas, indicando os caminhos possíveis. No mundo real, ruas contendo mãos de direção podem ser representadas por grafos direcionados. Esse tipo

de grafo é apresentado na próxima figura.

Neste exemplo, o grafo será representado pelos seguintes conjuntos G = ({1, 2, 3}, {(1, 1), (2, 1), (3, 1), (3, 2), (2, 3)}). Note que a tupla (3, 2) indica uma aresta indo do vértice 3 para o 2, enquanto a tupla (2, 3) indica uma aresta indo do vértice 2 para o 3. Note também que só existe uma única direção direta indo do vértice 2 para o 1, mas o sentido contrário não é possível. Essa aresta indica um caminho de mão única do ponto 2 ao ponto 1.

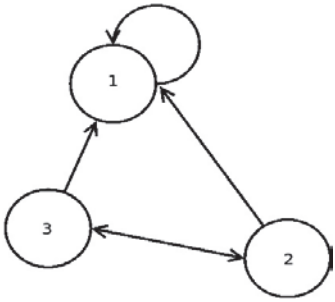


Figura 10.3: Exemplo de grafo direcionado

10.2 GERADOR DE LABIRINTO

O gerador de labirintos usado neste livro é baseado no conceito de busca em profundidade. O algoritmo escolhe aleatoriamente uma célula; em seguida, essa célula é marcada como visitada e escolhe-se, de forma aleatória, uma das células vizinhas. Se o vizinho não foi visitado ainda, é removido um muro entre essa célula e o seu vizinho. O algoritmo pula então para a célula vizinha, a qual se torna a célula corrente. Esse processo se repete até que todas as células tenham sido visitadas.

O labirinto será representado por uma matriz quadrada,

contendo um número ímpar de linhas e colunas. As células a serem visitadas ficarão nas posições i's e j's de valores ímpares. Isso facilitará a criação de um labirinto contendo um muro que contorna todo o desenho. Na figura a seguir é representada o mapa inicial, em que os pontos amarelos são células a serem visitadas.

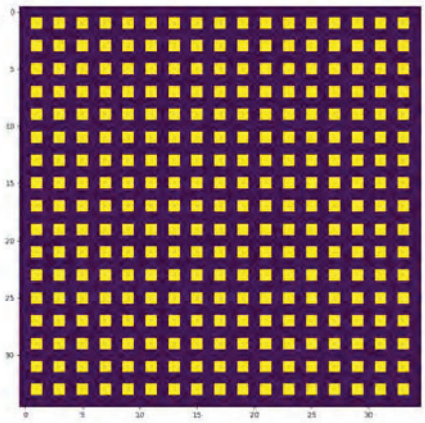


Figura 10.4: Mapa inicial para criação de labirinto para uma matriz de 35x35

Podemos pensar na busca em profundidade como o processo de subir em uma árvore, seguir em um galho até a ponta mais alta. Depois, vasculhamos as pontas mais baixas dos galhos. Em seguida, passamos para a ponta mais alta do próximo galho, e repetindo o processo anterior. Esses passos serão repetidos até que se visite todos os galhos da árvore, ou se chegue a um galho específico.

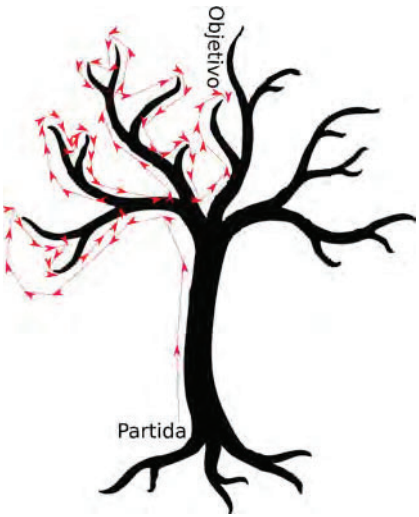


Figura 10.5: Exemplo de busca em profundidade em galhos de uma árvore.

Para entender melhor o algoritmo para geração de labirinto, podemos começar com o seguinte exemplo, considere a tabela a seguir:

0	**0**	**0**	**0**	**0**
0	1	0	1	0
0	0	0	0	0
0	1	0	1	0
0	0	0	0	0

Nesse caso, o número 1 representa a célula e o 0 representa o muro. Considere que foi escolhido primeiramente, a célula mais à esquerda e mais acima como ponto de partida. Em seguida foi escolhido o vizinho imediatamente à direita.

A próxima tabela representa a remoção do muro.

0	**0**	**0**	**0**	**0**
0	1	1	1	0
0	0	0	0	0
0	1	0	1	0
0	0	0	0	0

No próximo passo, a célula vizinha se torna a corrente. A próxima opção é a célula imediatamente abaixo, o que gera a seguinte tabela:

0	**0**	**0**	**0**	**0**
0	1	1	1	0
0	0	0	1	0
0	1	0	1	0
0	0	0	0	0

Esse processo se repete para a próxima célula vizinha que ainda não foi visitada, gerando a seguinte tabela final:

0	**0**	**0**	**0**	**0**
0	1	1	1	0
0	0	0	1	0
0	1	1	1	0
0	0	0	0	0

Como a próxima célula seria a do ponto de partida, o algoritmo para, pois essa célula já está armazenada na tabela de visitadas. A busca em profundidade ocorre ao visitarmos todos os vizinhos próximos de uma célula antes de seguir adiante.

Vamos criar um arquivo chamado `gerarlabirinto.py`, no qual usaremos os seguintes módulos:

```
from numpy import zeros, array
from random import shuffle, choice
```

A função `zeros` será usada para criar a matriz quadrada inicializada com valores numéricos iguais a zero. Nesse exemplo usaremos as funções aleatórias `shuffle`, para embaralhar listas, e a função `choice`, para escolher um valor, de forma aleatória, na lista.

Será criada a classe `GerarLabirinto`, cujo construtor é dado por:

```
class GerarLabirinto:
    "Gerador de Labirinto"
    def __init__(self, dimensao):
    dimensao = dimensao + 1 if dimensao % 2 == 0 else dimensao
        self._dimensao = dimensao
        self._mapa = zeros((dimensao, dimensao))

        celulas = array([(2*i +1, 2*j + 1) for i in range((dimens
ao -1) // 2)
                                           for j in range((dimens
ao -1) // 2)])
        celulas = (celulas[:,0], celulas[:,1])
        self._mapa[celulas] = 255
        self._celulas = list(zip(celulas[0], celulas[1]))
        self._visitados = []
        self._caminhar(*choice(self._celulas))
```

O construtor recebe como parâmetro a variável `dimensao`, que é o total de linhas/colunas da matriz quadrada, a qual representará o labirinto. Em seguida, usando uma expressão condicional, garantimos que a dimensão terá valor ímpar. O próximo passo é a criação inicial do mapa, em que todos os valores são zeros. A variável `celulas` conterá as posições ímpares de `i`

`e j` , que representaram as possíveis regiões a serem visitadas.

Note que a expressão matemática 2x+1 sempre retorna um número ímpar, x, pertencendo ao conjunto dos números inteiros. Para manipular a matriz `self._mapa` a partir dos índices gerados anteriormente, reescrevemos a variável `celulas` na forma de uma tupla, em que o primeiro elemento representa as linhas a serem usadas, enquanto o segundo fecha o par coluna. Com a declaração `self._mapa[celulas] = 255` estamos convertendo todos os dados no mapa, definidos pelas posições dadas pela variável `celula` em valor 255.

O passo seguinte é a criação de uma lista contendo pares ordenados `i,j` , os quais representaram as posições `x,y` das células a serem navegadas. A lista `_visitados` é inicializada como vazia, e depois conterá os pontos os quais o algoritmo já visitou. A última linha realiza a chamada do método `self._caminhar` o qual recebe um dos pares `x,y` , o qual será o ponto de partida.

> O operador `*` , ao ser aplicado no argumento de uma variável de uma função, expande a lista, ou tupla em variáveis de uma função. Por exemplo, seja a função `soma(a,b)` , que recebe dois parâmetros `a` e `b` . Essa função pode ser chamada da seguinte forma `soma(*[1,2])` . Nesse caso, o parâmetro `a` receberá o número 1 enquanto o parâmetro `b` receberá o número 2.

Em seguida, serão criados o método `gerar_novo_mapa` , que

permitirá reiniciar o mapa, e o método que permite o acesso ao atributo `self._mapa`:

```
@property
def mapa(self):
  return self._mapa

def gerar_novo_mapa(self):
  self._mapa = zeros((dimensao, dimensao))
  self._caminhar(*choice(self._celulas))
```

Um método auxiliar será criado, para verificar se um possível vizinho está dentro dos limites válidos do mapa:

```
def _cond(self, x, y):
    return (x >= 0 and x <= self._dimensao - 1 and
                y >= 0 and y <= self._dimensao - 1 )
```

O método retornará `True` se os valores de `x` e `y` estiverem nos limites do mapa.

O último método a ser apresentado é o `_caminhar`:

```
def _caminhar(self, x, y):
  vizinhos = [(x - 2, y), (x + 2, y), (x, y - 2),(x, y + 2)]
  vizinhos = [vi for vi in vizinhos if self._cond(*vi)]
  shuffle(vizinhos)
  self._visitados.append((x, y))
  for (xx, yy) in vizinhos:
    if (xx, yy) not in self._visitados:
      if x == xx:
        yp = y + 1 if y - yy < 0 else y - 1
        self._mapa[x, yp] = 255
      if y == yy:
        xp = x +1 if x - xx < 0 else x - 1
        self._mapa[xp, y] = 255
      self._caminhar(xx, yy)
```

O método `_caminhar` recebe como parâmetros a posição `x` e `y` da célula que está sendo visitada. O primeiro passo é o de montar a lista contendo as posições dos possíveis vizinhos. Em

seguida, usando o método `_cond`, realizamos a filtragem para que sobrem apenas os vizinhos que realmente estão dentro do mapa.

A função `shuffle` embaralha a lista de vizinhos. A posição `x,y` da célula atual é então armazenada na lista de visitados. Em seguida, começamos a percorrer os vizinhos próximos. Se o valor `xx, yy` da posição do vizinho não estiver na lista de visitados, então é verificado qual muro deve ser removido.

Note que, se `x` for igual a `xx`, isso indica que o vizinho está na mesma linha que a célula atual. Similarmente, se `y` for igual a `yy`, então a célula vizinha está na mesma coluna que a célula corrente.

Ao final, uma chamada recursiva do método `_caminhar` é executada, para que a busca continue descendo (busca em profundidade) pelos vizinhos próximos, até que todos os pontos próximos de uma região sejam visitados.

Na próxima figura é apresentado um exemplo de labirinto gerado por esse algoritmo:

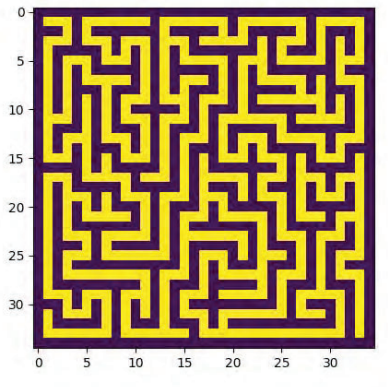


Figura 10.6: Exemplo de labirinto de 35x35 gerado pelo algoritmo de busca em profundidade

O código completo do módulo `gerarlabirinto.py` é apresentado a seguir:

```
#!/usr/bin/env python3.6
# -*- Coding: UTF-8 -*-
"""
Gerador de Labirinto.
"""
from numpy import zeros, array
from random import shuffle, choice


class GerarLabirinto:
    "Gerador de Labirinto"
    def __init__(self, dimensao):
    dimensao = dimensao + 1 if dimensao % 2 == 0 else dimensao
    self._dimensao = dimensao
        self._mapa = zeros((dimensao, dimensao))
        celulas = array([(2*i +1, 2*j + 1) for i in range((dimens
ao -1) // 2)
                                           for j in range((dimens
ao -1) // 2)])
        celulas = (celulas[:,0], celulas[:,1])
        self._mapa[celulas] = 255
        self._celulas = list(zip(celulas[0], celulas[1]))
        self._visitados = []
        self._caminhar(*choice(self._celulas))

    @property
    def mapa(self):
        return self._mapa

    def gerar_novo_mapa(self):
    self._mapa = zeros((dimensao, dimensao))
        self._caminhar(*choice(self._celulas))

  def _cond(self, x, y):
          return (x >= 0 and x <= self._dimensao - 1 and
                  y >= 0 and y <= self._dimensao - 1 )

    def _caminhar(self, x, y):
        vizinhos = [(x - 2, y), (x + 2, y), (x, y - 2),(x, y + 2)
]
        vizinhos = [vi for vi in vizinhos if self._cond(*vi)]
```

```
        shuffle(vizinhos)
        self._visitados.append((x, y))
        for (xx, yy) in vizinhos:
    if (xx, yy) not in self._visitados:
        if x == xx:
            yp = y + 1 if y - yy < 0 else y -1
            self._mapa[x, yp] = 255
        if y == yy:
            xp = x +1 if x - xx < 0 else x - 1
            self._mapa[xp, y] = 255
        self._caminhar(xx, yy)
```

Conclusão

Neste capítulo foi apresentada uma breve introdução ao conceito de grafos e ao algoritmo de geração de labirintos baseado em busca em profundidade. No próximo capítulo apresentaremos como converter a matriz que representa o labirinto em um grafo conectado. Também será implementado o algoritmo evolucionário que encontra o caminho entre dois pontos dentro do labirinto.

Capítulo 11

DA IMAGEM AO GRAFO

No capítulo anterior, foram apresentados conceitos de grafos e o algoritmo para geração de labirintos. No intuito de criar um algoritmo generalizado para solucionar labirintos, vamos desenvolver uma classe para navegar através da matriz que representa os caminhos possíveis.

O interessante desse estudo de caso é que podemos pensar em ruas de uma cidade como sendo caminhos de um labirinto. Como exemplo, na próxima imagem é apresentado um recorte de um mapa de um bairro da cidade de São José dos Campos. Nesse caso, estamos interessados em sair do ponto vermelho, na R. Orion, e chegar ao ponto azul na R. Sagitarios.

Figura 11.1: Trecho de mapa de um bairro de São José dos Campos. Ponto vermelho sendo a partida e o azul a chegada.

Baseado nesse caso de estudo, podemos pensar na aplicação de computação evolucionária a sistemas autônomos de navegação, seja por robôs, carros ou mesmo em veículos aéreos não tripulados.

11.1 DO PIXEL AO VÉRTICE

Imagens coloridas podem ser representadas pela composição das camadas RGB (Vermelha - *Red*; Verde - *Green*; Azul - *Blue*). Nesse caso, teríamos uma matriz de dimensão 3 x W x H, em que W representa a largura, em pixel, da imagem e H a sua altura. Ou seja, temos três matrizes de W x H formando um cubo de dados. Já uma imagem em tons de cinza é simplesmente uma matriz W x H, em que cada ponto tem-se valores que vão de 0 a 255. O valor 0 representa preto e o 255, branco, valores intermediários são tons

de cinza. Assim, uma imagem nada mais é que uma matriz ou vetor multidimensional.

O algoritmo apresentado no capítulo anterior gera uma matriz quadrada, em que o muro tem valor 0 e o caminho 255. Dessa forma, o labirinto gerado é uma imagem preto e branco. O primeiro passo será o de converter pixel, que representam caminhos, em vértices do grafo. As arestas serão formadas pelas posições dos vizinhos imediatamente próximos a um dado ponto.

Vamos utilizar a biblioteca *NetworkX*, a qual facilita a manipulação de grafos. A instalação dessa biblioteca pode ser realizada utilizando o comando:

```
pip install networkx
```

O próximo passo será o de criar o arquivo `labgrafo.py`, no qual vamos importar os seguintes módulos:

```
from numpy import array, where
import networkx as nx
```

A maior parte dos módulos importados já foram usados previamente ao longo do livro, a novidade fica para o uso do `networkx`.

Vamos criar a classe `LabGrafo`, cujo construtor é apresentado no código a seguir:

```
class LabGrafo:
    def __init__(self, img):
        self._img = img
        self._nos, self._nome_nos, self._grafo = self._criar_graf
o()
```

Observe que a classe recebe a variável `img`, que será a matriz

representando a imagem do labirinto. Note também que usamos o método `_criar_grafo` para gerar os atributos `_nos`, que representa os nós do grafo, `_nome_nos`, o qual contém a representação do nome dos nós e o atributo `_grafo` será o grafo em si, construído a partir da biblioteca `NetworkX`.

O método `_criar_grafo` é dado por:

```
def _criar_grafo(self):
    '''
    Gera grafo a partir da matriz imagem, que representa o labiri
nto.
    Valores iguais a 0 indicam muro.
    '''
    caminhos = where(self._img != 0)
    nos = array(list(zip(caminhos[0], caminhos[1])))
    nnos = nos.shape[0]
    vertices = LabGrafo.encontrar_vertices(nos)
    grafo = nx.Graph()
    nome_nos = list(range(nnos - 1))
    grafo.add_nodes_from(nome_nos)
    grafo.add_edges_from(vertices)
    return nos, nome_nos, grafo
```

Na primeira declaração, `caminhos = where(self._img != 0)`, obtemos todas as posições `i,j` na matriz que representa o labirinto, para os valores que são diferentes de zero. Ou seja, essa variável conterá todos os pontos que formam caminhos no labirinto, desprezando os muros. Em seguida, geramos um vetor contendo pares ordenados dos pontos `i,j`. Cada par ordenado nessa lista representará um nó no grafo. A variável `nnos` armazena o total de nós que vão formar o nosso grafo.

Usando o método estático `encontrar_vertices`, são geradas as relações de vértices possíveis formados a partir de caminhos do labirinto. Com a declaração `grafo = nx.Graph()`, inicializamos um novo grafo, usando o pacote `networkx`. Observe que o nome

dos nós, dado pela variável `nome_nos`, será um número inteiro, definido pela posição do par ordenado no vetor `nos`. Nas duas declarações finais, antes do retorno, adicionamos os nomes dos nós e os vértices ao grafo. O método retorna os nós, nome dos nós e o grafo.

Um método estático é aquele que não possui acesso direto aos atributos e métodos da classe, agindo como uma função independente. Além disso, tais métodos são utilizados sem a necessidade instanciar um objeto da classe. Para isso, utiliza-se o nome da classe mãe, seguido por um ponto e o método estático, por exemplo, `LabGrafo.encontrar_vertices(nos)`.

No trecho de código a seguir é apresentado o método `encontrar_vertices`.

```
@staticmethod
def encontrar_vertices(nos):
    """
    Encontra os vértices nas regiões de movimentos permitidos ent
re os nós.
    """
    n = nos.shape[0]
    vertices = []
    for i in range(0, n - 1):
        for j in range(i, n):
            if abs(nos[i, 0] - nos[j, 0]) > 1:
                break
            elif abs(nos[i, 0] - nos[j, 0]) == 1 and \
                 abs(nos[i, 1] - nos[j, 1]) == 0:
                 vertices.append([i, j])
            elif abs(nos[i, 0] - nos[j, 0]) == 0 and \
                 abs(nos[i, 1] - nos[j, 1]) == 1:
                 vertices.append([i, j])
    return array(vertices)
```

Observe que foi utilizado o decorador `@staticmethod` para indicar que o método abaixo é estático. Vale notar também que tal

método não recebe o `self` como parâmetro.

No algoritmo `encontrar_vertices`, `n` contém o número de nós. A variável `vertices` é inicializada como uma lista vazia. Os laços em `i` e `j` são usados para varrer os pontos dos nós. Se um ponto for vizinho imediato a outro, calculado pela diferença absoluta entre as posições relativas dos pontos `(i, j)`, esses serão adicionados à lista de vértices. Ao final, é retornado um vetor que representa os vértices que podem ser gerados a partir dos pares ordenados que formam os nós do grafo. Dessa forma, estamos convertendo pixels de caminhos do labirinto em um grafo conectado.

Usamos o decorador `property` para acessar os atributos da classe:

```
@property
def nos(self):
    '''Retorna os nomes dos nós.'''
    return self._nome_nos

@property
def nos_ij(self):
    '''
    Retorna o vetor contendo o par que relaciona posição com nó
    na matriz da imagem de entrada.
    '''
    return self._nos

@property
def grafo(self):
    '''
    Retorna o grafo que representa os caminhos possíveis no labir
into.
    '''
    return self._grafo
```

A seguir, é apresentado o código completo contido no módulo

labgrafo.py .

```
#!/usr/bin/env python3.6
# -*- Coding: UTF-8 -*-
'''
Converte Labirinto em um grafo.
'''
from numpy import array, where
import networkx as nx


class LabGrafo:
    def __init__(self, img):
      self._img = img
      self._nos, self._nome_nos, self._grafo = self._criar_grafo(
)

    def _criar_grafo(self):
        '''
        Gera grafo a partir da matriz imagem, que representa o la
birinto.
        Valores iguais a 0 indicam muro.
        '''
        caminhos = where(self._img != 0)
        nos = list(zip(caminhos[0], caminhos[1]))
        nnos = len(nos)
        nos = array(nos)
        vertices = LabGrafo.encontrar_vertices(nos)
        grafo = nx.Graph()
        nome_nos = list(range(nnos - 1))
        grafo.add_nodes_from(nome_nos)
        grafo.add_edges_from(vertices)
        return nos, nome_nos, grafo

    @property
    def nos(self):
        '''Retorna os nomes dos nós.'''
        return self._nome_nos

    @property
    def nos_ij(self):
        '''
        Retorna o vetor contendo o par que relaciona posição com
nó
```

```
        na matriz da imagem de entrada.
        '''
        return self._nos

    @property
    def grafo(self):
        '''
        Retorna o grafo que representa os caminhos possíveis no l
abirinto.
        '''
        return self._grafo

    @staticmethod
    def encontrar_vertices(nos):
        """
        Encontra os vértices nas regiões de movimentos permitidos
 entre os nós.
        """
        n = nos.shape[0]
        vertices = []
        for i in range(0, n - 1):
            for j in range(i, n):
                if abs(nos[i, 0] - nos[j, 0]) > 1:
                    break
                elif abs(nos[i, 0] - nos[j, 0]) == 1 and \
                     abs(nos[i, 1] - nos[j, 1]) == 0:
                     vertices.append([i, j])
                elif abs(nos[i, 0] - nos[j, 0]) == 0 and \
                     abs(nos[i, 1] - nos[j, 1]) == 1:
                     vertices.append([i, j])
        return array(vertices)
```

Conclusão

Neste capítulo, foi criada a classe base que converter uma matriz, a qual representa um labirinto, em um grafo conectado. Para alcançar esse objetivo, ficou definido que pontos na matriz com valor 0 representam muro, enquanto valores 255 indica caminhos. A base da conversão de labirinto para grafos consistiu em montar o vetor de nós e vértices, considerando pixels que eram

vizinhos imediatos.

No próximo capítulo será criada uma classe, a qual estenderá `LabGrafo` , que terá o método de avaliação a ser usado pelo algoritmo genético, para determinar o melhor caminho de um ponto A ao B no labirinto.

Capítulo 12

ENCONTRANDO CAMINHOS

Quando queremos resolver um problema, a grande dificuldade está justamente em como modelá-lo para ser solucionado através do computador. Para o labirinto, primeiramente, foi necessário converter o problema de caminhar pelos corredores em navegação por nós de um grafo conectado. Nesse caso, foi aplicado o conceito de redutibilidade de problemas. O objetivo final é o de encontrar o caminho mais curto entre um ponto A e B. Esse é um problema de otimização, em que se deseja achar a melhor solução em um universo de possibilidades. Para alcançar esse objetivo, vamos usar algoritmo genético, em que a codificação genética de cada indivíduo vai representar probabilidades de se escolher seguir em uma direção ou outra.

12.1 AVALIAÇÃO DE INDIVÍDUO

Uma vez que desenvolvemos o sistema que permite navegar através dos corredores do labirinto, representado por grafos conectados, o próximo passo será o de escrever a função de avaliação para ponderar se um dado caminho é viável ou não. Aqui vamos estender a classe `LabGrafo` para realizar essa tarefa.

Vamos criar o arquivo chamado `labgrafogen.py`, no qual vamos realizar a seguinte importação:

```
from labgrafo import LabGrafo
```

O construtor da classe `LabGrafoGen` é dado por:

```
class LabGrafoGen(LabGrafo):
    '''
    Estende a classe de representação de labirinto em grafo para
adicionar
    avaliação de indivíduo vindo do algoritmo genético.
    '''
    def __init__(self, img):
        super(LabGrafoGen, self).__init__(img)
```

Note que aqui simplesmente inicializamos a classe mãe, usando o método `super`, a qual recebe o parâmetro `img`. A função de avaliação será dada pelo seguinte código:

```
def avaliacao(self, individuo, inicio, meta, upcaminho=True):
      '''
      Dado um indivíduo, verifica se o mesmo representa uma soluç
ão
      para o labirinto.
      '''
      partida = self._no_correspondente(inicio)
      fim = self._no_correspondente(meta)
      caminho = [partida]
      atual = partida
      chegou = False
      fitness = 0
      i0 = 0

      while True:
          possibilidades = {ng for ng in  self._grafo.neighbors(a
tual)}
          possibilidades = list(possibilidades - set(caminho))
          nposs = len(possibilidades)
          if nposs > 1:
              genes = individuo[i0: i0 + nposs]
              escolhido = possibilidades[where(genes == max(genes
```

```
))[0][0]]
                i0 += nposs
                caminho.append(escolhido)
                atual = escolhido
            elif nposs == 1:
                caminho += possibilidades
                atual = possibilidades[0]
            else:
                break

            if atual == fim:
                chegou = True
                break

        if chegou is False:
            fitness = len(caminho) ** 2.0  / 1e9
        else:
            fitness = 1.0 / len(caminho)

        if upcaminho is True:
            self._caminho = caminho

        return fitness
```

Antes da declaração `While True` , estamos inicializando as variáveis que definem os pontos, de partida, `partida` , e final, `fim` . A variável que vai armazenar os caminhos percorridos, usando como base o código genético de um indivíduo, é dada pela lista `caminho` . Também inicializamos a variável `atual` , que armazenará o ponto corrente percorrido no labirinto. A variável `chegou` será usada como condição de parada para finalizar o laço infinito (verdadeiro se alcançou o objetivo). A variável total representará o número de passos que foram dados para chegar até o objetivo. Já a variável `i0` será um contador, usada para mapear a cadeia genética do indivíduo.

A declaração `possibilidades = {ng for ng in self._grafo.neighbors(atual)}` cria um conjunto (*set*)

contendo os vizinhos do nó atual. Com a declaração `possibilidades = list(possibilidades - set(caminho))` as possibilidades passam a ser uma lista contendo a diferença entre o conjunto de nós vizinhos menos o conjunto de nós já percorridos (diferença de conjuntos). Com isso, evitamos que o nosso navegador faça uma trajetória repetida pelo labirinto. As possibilidades sempre serão os caminhos ainda não percorridos.

Observe que, quando o agente percorre o labirinto, ele poderá chegar a um corredor sem saída. Se isso ocorrer, o tamanho da lista de possibilidades será zero, e a estrutura condicional `else` será chamada, encerrando o laço de repetição com o comando `break`. Nesse caso, a variável `chegou` passa a ter valor `False`.

Outra possibilidade são os pontos de cruzamento, em que existe mais de uma direção possível para seguir. Nesses pontos, usaremos a codificação genética do indivíduo para tomar a decisão de qual direção seguir. Inspirado na engenharia genética, fatiamos trechos do código genético, tal como seria feito por uma enzima. O tamanho da fatia será dado pelo número total de pontos que podemos escolher no cruzamento, `genes = individuo[i0: i0 + nposs]`. O nó escolhido será aquele gene com o maior valor numérico. Dessa forma, os genes representarão probabilidades de escolher um determinado caminho. Aqui vale ressaltar que o código `where(genes == max(genes)[0][0]` retorna a posição do primeiro gene igual ao máximo na lista. Esse ponto é então adicionado à lista de caminhos percorridos.

Por outro lado, se a única possibilidade for a seguir em frente (lista de possibilidades tem apenas um elemento), não usamos a codificação genética para essa decisão, simplesmente

incrementamos a lista a `caminho`. Nesses dois casos, a variável `atual` é atualizada com a posição corrente no labirinto. Ao final, se não foi alcançada a posição objetivo dentro do labirinto, adicionamos uma penalidade ao valor da variável `fitness`, a qual será proporcional ao total de passos percorridos, dividido por um bilhão. Ao se chegar ao objetivo, o valor de `fitness` será dado pelo inverso do número de passos.

Com isso, os objetivos serão os seguintes: se não for possível chegar ao ponto de interesse, os melhores indivíduos serão aqueles que mais avançaram dentro do labirinto. Caso o ponto de interesse seja alcançado, serão considerados como indivíduos mais aptos aqueles que chegam ao final com o menor número de passos.

Para visualizar o processo de evolução, para esse problema, criamos o método `plot`, o qual será utilizado para gerar uma animação do caminho percorrido por um agente, que utiliza como decisor a codificação genética de um dado indivíduo.

```
def plot(self, individuo, inicio, meta, geracao=None,
         save_file=None, interval=100):
      '''
      Gera um vídeo do percurso definido pelos genes de um dado i
ndivíduo.
      '''
      self.avaliacao(individuo, inicio, meta)

      i_fim, j_fim = meta

      fig, ax = plt.subplots()
      img = self._img.copy()

      img[i_fim, j_fim] = 200
      img[img == 0] = 255
      img[img == -1] = 0
      img[inicio] = 0
```

```
        im = ax.imshow(img, animated=True, interpolation='none', as
pect='auto')

        def updatefig(frame):
            img = self._img.copy()
            img[inicio] = 255
            img[i_fim, j_fim] = 200
            atl = self._nos[self.caminho[frame]]
            img[atl[0], atl[1]] = 100
            if geracao is not None:
                label = "Geração {}".format(geracao)
                ax.set_title(label)
            im.set_array(img)
            return im,

        ani = FuncAnimation(fig,
                            updatefig,
                            frames=len(self.caminho),
                            interval=interval,
                            blit=False)

        if save_file is not None:
            Writer = writers['ffmpeg']
            writer = Writer(fps=29, metadata=dict(artist='Me'), bit
rate=1800)
            ani.save(save_file, writer=writer)
        else:
            plt.show()
```

O método recebe como parâmetros a codificação genética de um indivíduo, o ponto de partida, `inicio`, o ponto de chegada, `meta`. Os argumentos com valores padrão são a geração atual, `geracao=None`, o nome do vídeo a ser salvo, `save_file=None`, e o intervalo (número de *frames*) a ser usado para gerar a animação, `interval=100`.

A primeira coisa realizada no algoritmo do método anterior é obter o caminho percorrido, a partir de um ponto, usando o método `avaliacao`. Em seguida, separamos o valor de i e j que representam o ponto de interesse. O próximo passo consiste na

inicialização da figura, `fig, ax = plt.subplots()`, a qual usaremos para gerar a animação. Uma cópia da imagem original do labirinto é feita, e definimos cores diferentes para o ponto de chegada (`img[i_fim, j_fim] = 200`). A variável `im` vai armazenar a referência para o objeto que manipula a exibição da matriz na forma de imagem.

Observe que no `imshow` indicamos que vamos criar uma animação (`animated=True`). Também é indicado que não se quer realizar uma interpolação (`interpolation='none'`), pois queremos que cada pixel no labirinto seja exibido como um pixel individual na imagem. Por fim, indicamos que o *matplotlib* deve ajustar de forma automática as dimensões da imagem (`aspect='auto'`). A função `updatefig` será utilizada para atualizar o gráfico a cada novo *frame*. Para cada *frame* é feita uma cópia da imagem do labirinto. O ponto de partida fica na cor branca `255` , enquanto o ponto de chegada fica em um tom de cinza `200` . Já o tom de cinza de valor 100 vai representar o agente que se move pelo labirinto.

Caso o parâmetro `geracao` não seja nulo, o título do gráfico é atualizado para conter a informação sobre qual a geração do indivíduo utilizado para criar o vídeo. A declaração `im.set_array(img)` atualiza os dados do gráfico, utilizando a imagem mais recente, armazenada em `img` .

Tal como feito em capítulos anteriores do livro, usamos a função `FuncAnimation` para gerar a animação. Se a variável `save_file` não for nula, então o vídeo será salvo em um arquivo local. Para isso, será utilizado a ferramenta **ffmpeg** (`Writer = writers['ffmpeg']`), com as configurações dadas pela declaração

`writer = Writer(fps=29, metadata=dict(artist='Me'), bitrate=1800)`. O vídeo será salvo com 29 *frames* por segundo e taxa de escrita de 1800. O método `ani.save(save_file, writer=writer)` salva o arquivo de vídeo com as propriedades definidas pelo `writer`.

> **FFmpeg**: programa livre, em linha de comando, que grava, converte e cria stream de áudio e vídeo em diversos formatos. Disponível para diversos sistemas operacionais e pode ser baixado em https://www.ffmpeg.org/.

Agora que temos uma maneira de mapear código genético em decisores de caminhos, vamos implementar o algoritmo evolucionário responsável por produzir indivíduos que representem a solução ótima do problema.

12.2 ALGORITMO GENÉTICO

Vamos criar um arquivo chamado `main.py`, no qual importaremos as bibliotecas necessárias para o projeto:

```
from numpy import load, save, where, array, random, hsplit, concatenate

import matplotlib.pyplot as plt

from pygenec.populacao import Populacao
from pygenec.selecao.torneio import Torneio
from pygenec.cruzamento.embaralhamento import Embaralhamento
from pygenec.mutacao.sequenciareversa import SequenciaReversa
from pygenec.evolucao import Evolucao
from pygenec import binarray2int
```

```
from gerarlabirinto import GerarLabirinto
from labgrafogen import LabGrafoGen
```

Os módulos e funções utilizados aqui já foram apresentados previamente ao longo do livro.

O script será iniciado com a opção de salvar um labirinto ou carregar um mapa salvo previamente:

```
car_salv = input("Carregar ou Salvar [c/s]: ")
filname = "./labirinto_grap.npy"

if car_salv == "s":
    dimensao = int(input("Qual a dimensão do Labirinto? "))
    lab = GerarLabirinto(dimensao)
    mapa = lab.mapa
    save(filname, mapa)
else:
    mapa = load(filname)


meta = (mapa.shape[0]- 2, mapa.shape[1]-5)
ponto_de_partida = (1, 1)

labgrafo = LabGrafoGen(mapa)
```

Se a opção for a de salvar um labirinto (`car_salv == "s"`), então a classe `GerarLabirinto` é chamada, usando as dimensões do labirinto, digitadas previamente pelo usuário. O mapa é armazenado na variável `mapa` , sendo salvo em um arquivo no formato de vetor do numpy (`.npy`). A variável `meta` define a meta, ou seja, o ponto de interesse, ao qual queremos chegar, já a variável `ponto_de_partida` indica a posição inicial no labirinto. O objeto `labgrafo` contém a representação do labirinto na forma de um grafo conectado.

As variáveis a seguir vão armazenar informações sobre os

atributos do algoritmo genético, como tamanho da população, total de cromossomos, tamanho da subpopulação, usada no cruzamento, número de bits por cromossomo, carga genética total, probabilidades de mutação e cruzamento, qual geração poderá ocorrer epidemia e se o algoritmo usará estratégia elitista.

```
tamanho_populacao = 50
cromossomos = len(labgrafo.nos)
tamanho = int(0.1 * tamanho_populacao)
bits = 4
genes = bits * cromossomos
pmut = 0.1
pcruz = 0.6
epidemia = 50
elitista = True
```

A função a seguir será utilizada para converter os cromossomos binários em valores numéricos de ponto flutuante.

```
def valores(populacao):
    bx = hsplit(populacao, cromossomos)
    const = 2 ** bits - 1
    const = 100 / const
    x = [const * binarray2int(xi) for xi in bx]
    x = concatenate(x).T.astype(int)
    return x
```

Essa função foi apresentada previamente. A diferença com relação ao mostrado em outros capítulos é que, aqui, um cromossomo poderá ter valor entre 0 e 100, definido pela declaração `const = 100 / const` .

Precisamos de uma função que receba a população de uma dada geração e realize a avaliação, definindo quais os indivíduos representam uma melhor solução para o problema.

```
def avaliacao(populacao):
    x = valores(populacao)
    n = len(populacao)
```

```
    def steps(k):
        individuo = x[k, :]
        t =  labgrafo.avaliacao(individuo, ponto_de_partida,
                                meta, upcaminho=False)
        return t

    peso = array([steps(k) for k in range(n)])
    return peso
```

Note que, dentro da subfunção, chamada `steps`, a variável `t` utiliza o método `avaliacao` do objeto `labgrafo`.

Em seguida, serão criadas instâncias da população, dos operadores genéticos e do operador de evolução. Neste capítulo serão utilizados o torneio como método de seleção, embaralhamento para o cruzamento e sequência reversa para a mutação. Um exercício interessante para o leitor é o de testar diferentes operadores genéticos, assim como modificações dos parâmetros do algoritmo, para verificar se existe uma opção que acelere o processo de busca.

```
populacao = Populacao(avaliacao,
                          genes,
                          tamanho_populacao)

selecao = Torneio(populacao, tamanho=tamanho)
cruzamento = Embaralhamento(tamanho_populacao)
mutacao = SequenciaReversa(pmut=pmut)

evolucao = Evolucao(populacao,
                        selecao,
                        cruzamento,
                        mutacao)

evolucao.nsele = tamanho
evolucao.pcruz = pcruz
evolucao.manter_melhor = elitista
evolucao.epidemia = epidemia
```

Aqui, será definido que a evolução vai ocorrer por no máximo duzentas gerações (`range(200)`):

```
for i in range(200):
    vmin, vmax = evolucao.evoluir()

    print(evolucao.geracao, vmax, vmin)
    if evolucao.geracao % 50 == 0:
        x = valores(populacao.populacao)
        individuo = x[-1, :]
        print("Gerando Video")
        fnome = "./videos/lab_{}.mp4".format(evolucao.geracao)
        labgrafo.plot(individuo, ponto_de_partida, meta,
                      geracao=evolucao.geracao, save_file=fnome)
    if vmax >=  0.002:
        break
```

Em cada geração será coletada a informação de mínimo e máximo. A cada 50 gerações será gerada uma animação, usando o melhor indivíduo. Se durante a evolução o valor da variável `vmax` for maior ou igual a 0.002, o processo será interrompido, pois teremos encontrado uma solução.

Sempre que o laço responsável pela evolução finalizar, será gerada uma animação da melhor solução encontrada:

```
x = valores(populacao.populacao)
individuo = x[-1, :]
print("Gerando Video")
fnome = "./videos/lab_{}.mp4".format(evolucao.geracao)
labgrafo.plot(individuo, startpoint, goal,
              geracao=evolucao.geracao, save_file=fnome)
```

O algoritmo evolucionário completo é apresentado a seguir:

```
from numpy import load, save, where, array, random, hsplit, conca
tenate

import matplotlib.pyplot as plt

from pygenec.populacao import Populacao
```

```
from pygenec.selecao.torneio import Torneio
from pygenec.cruzamento.embaralhamento import Embaralhamento
from pygenec.mutacao.sequenciareversa import SequenciaReversa
from pygenec.evolucao import Evolucao
from pygenec import binarray2int

from gerarlabirinto import GerarLabirinto
from labgrafogen import LabGrafoGen

car_salv = input("Carregar ou Salvar [c/s]: ")
filname = "./labirinto_grap.npy"

if car_salv == "s":
    dimensao = int(input("Qual a dimensão do Labirinto? "))
    lab = GerarLabirinto(dimensao)
    mapa = lab.mapa
    save(filname, mapa)
else:
    mapa = load(filname)


meta = (mapa.shape[0]- 2, mapa.shape[1]-5)
ponto_de_partida = (1, 1)

labgrafo = LabGrafoGen(mapa)

tamanho_populacao = 50
cromossomos = len(labgrafo.nos)
tamanho = int(0.1 * tamanho_populacao)
bits = 4
genes = bits * cromossomos
pmut = 0.1
pcruz = 0.6
epidemia = 50
elitista = True

def valores(populacao):
    bx = hsplit(populacao, cromossomos)
    const = 2 ** bits - 1
    const = 100 / const
    x = [const * binarray2int(xi) for xi in bx]
    x = concatenate(x).T.astype(int)
    return x

def avaliacao(populacao):
```

```
  x = valores(populacao)
  n = len(populacao)

  def steps(k):
      individuo = x[k, :]
      t =  labgrafo.avaliacao(individuo,
                              ponto_de_partida,
                              meta, upcaminho=False)
      return t

  peso = array([steps(k) for k in range(n)])
  return peso

populacao = Populacao(avaliacao,
                      genes,
                      tamanho_populacao)

selecao = Torneio(populacao, tamanho=tamanho)
cruzamento = Embaralhamento(tamanho_populacao)
mutacao = SequenciaReversa(pmut=pmut)

evolucao = Evolucao(populacao,
                    selecao,
                    cruzamento,
                    mutacao)

evolucao.nsele = tamanho
evolucao.pcruz = pcruz
evolucao.manter_melhor = elitista
evolucao.epidemia = epidemia
for i in range(200):
    vmin, vmax = evolucao.evoluir()

    print(evolucao.geracao, vmax, vmin)
    if evolucao.geracao % 50 == 0:
        x = valores(populacao.populacao)
        individuo = x[-1, :]
        print("Gerando Video")
        fnome = "./videos/lab_{}.mp4".format(evolucao.geracao)
        labgrafo.plot(individuo, ponto_de_partida, meta,
                      geracao=evolucao.geracao, save_file=fnome)
    if vmax >=  0.002:
        break
```

```
x = valores(populacao.populacao)
individuo = x[-1, :]
print("Gerando Video")
fnome = "./videos/lab_{}.mp4".format(evolucao.geracao)
labgrafo.plot(individuo, startpoint, goal,
              geracao=evolucao.geracao, save_file=fnome)
```

Conclusão

Neste capítulo, foi apresentada uma maneira de utilizar representação de uma cadeia genética como decisor sobre caminhos a seguir dentro de um grafo. Além disso, vimos que o problema de sair de um ponto a outro no labirinto pode ser tratado como encontrar o caminho mínimo dentro de um grafo. Esse tipo de problema é o similar aos que devem ser solucionados por algoritmos que calculam rotas em um mapa.

Capítulo 13

CONCLUSÕES

Algoritmos evolucionários são do tipo estocástico, empregados principalmente em problemas difíceis de resolver. Neste livro foi apresentado o procedimento de criação de um framework de computação evolucionária de propósito geral, chamado de *pygenec*. No processo de criação do framework foram apresentados os principais operadores genéticos de seleção, cruzamento, mutação e evolução. Ao longo do livro utilizamos de forma massiva a biblioteca numpy, que é a principal ferramenta, em Python, para manipulação de vetores e computação numérica.

Assim o livro procurou ser não somente uma referência sobre algoritmos genéticos como também uma ferramenta didática para que o leitor possa adquirir conhecimentos mais profundos sobre Programação Orientada a Objetos, métodos numéricos e vetoriais e aplicação de conceitos fundamentais de Ciência da Computação, como redutibilidade de problemas e teoria de grafos. Tudo de forma prática e contextualizada a problemas com aplicações reais.

13.1 REFERÊNCIAS BIBLIOGRÁFICAS

CHIWIACOWSKY, Leonardo D. e VELHO, Haroldo F. de. Different approaches for the solution of a backward heat

conduction problem. *Inverse Problems in Engng*. 11. 6. p. 471. 2003.

DARWIN, Charles. *On the origin of species*, 1859. Routledge, 2004.

GANGULY, Supriyo. *Solving Sudoku with a simple algorithm*. OpenSourceforu. 2018. https://opensourceforu.com/2018/11/solving-sudoku-with-a-simple-algorithm/ acessado em 01 de fevereiro de 2018.

GWIAZDA, Tomasz D. *Genetic algorithms reference*. v 1. http://www.tomaszgwiazda.com/wstep.htm acessado em 24 de janeiro de 2019.

FELGENHAUER, Bertram; JARVIS, Frazer. Mathematics of sudoku I. *Mathematical Spectrum*, v. 39, n. 1, p. 15-22, 2006.

HO, Yung Shwen, TANG, Vicente e CHIN, Leonard. *Genetic Algorithm*. http://www.cse.unsw.edu.au/~cs9417ml/GA2/crossover_alternate.html acessado em 25 de janeiro de 2019.

HOLLAND, John Henry et al. *Adaptation in natural and artificial systems*. Ann Arbor: University of Michigan Press, 1975.

JEBARI, Khalid e MADIAFI, Mohammend. *Selection Methods for Genetic Algorithms*. Int. J. Emerg. Scie. 3. p.333. 2013.

KIUSALAAS, Jaan. *Numerical methods in engineering with Python*. Cambridge University Press. Nova York. 433 p. 2005

LYNCE, Inês; OUAKNINE, Joël. *Sudoku as a SAT Problem*. In: ISAIM. 2006.

MANTERE, Timo; KOLJONEN, Janne. Solving, rating and generating Sudoku puzzles with GA. In: *Evolutionary Computation*, 2007. CEC 2007. IEEE Congress on. IEEE, 2007. p. 1382-1389.

NETO, Antônio J. D. S. BECCENERI, José C. VELHO, Haroldo F. D. C. *Inteligência Computacional aplicada a problemas inversos em transferência radiativa*. EdUERJ. Rio de Janeiro. 2016.

NISHA, Saine. *Review of Selection Methods in Genetic Algorithms*. Int. Jour. of Eng. and Comp. Scie. 6. 12. p.22261. 2017.

OLIVEIRA, Ricardo T. *Reduções de problemas em grafos com soluções conexas para (max)SAT e adaptação de um resolvedor SAT e MAXSAT não clausal para as instâncias obtidas*. Dissertação (mestrado) – Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática. Curitiba, 2013.

PACHECO, Marco A. C. et al. *Algoritmos genéticos: princípios e aplicações. ICA: Laboratório de Inteligência Computacional Aplicada*. Departamento de Engenharia Elétrica. Pontifícia Universidade Católica do Rio de Janeiro. http://www.inf.ufsc.br/~mauro.roisenberg/ine5377/Cursos-ICA/CE-intro_apost.pdf acessado em 22 de julho de 2019, p. 28, 1999.

PEREIRA, Eduardo S. *Trilhas Python*: Programação multiparadigma e desenvolvimento web com Flask. Casa do código. São Paulo. 2018.

SILVA, Ivan Nunes, SPATTI, Danilo Hernane, FLAUZINO, Rogério Andrade. *Redes Neurais Artificiais para engenharia e*

ciências aplicadas: fundamentos teóricos e aspectos práticos. 2. 431 p. 2016.

SIPSER, Michael. *Introdução à teoria da computação*. Cenage Learning. São Paulo. 459 p. 2. 2015.

SOUZA, Sandro Ferreira. *Estudo das propriedades e robustez da rede de transporte público de São Paulo*. Dissertação (Mestrado em Ciências) - Programa de Pós-Graduação em Modelagem de Sistemas Complexos, Escola de Artes, Ciências e Humanidades, Universidade de São Paulo. São Paulo. 2016.

TURING, Alan Mathison. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society*, v. 2, n. 1, p. 230-265, 1937.

UMBARKAR, Anantkumar J. e SHETH, P. D. *Crossover operators in genetic algorithms*: A review. ICTACT Jour. on Soft. Computing. 6. 1. p. 1083. 2015.

WEISS, John M. Genetic Algorithms and Sudoku. In: *Midwest Instruction and Computing Symposium* (MICS 2009). 2009. p. 1-9.

WIRTH, Elias. *How Google Maps Calculates the shortest route*. Disponível em https://mathsection.com/how-google-maps-calculates-the-shortest-route, acessado em 04 de agosto de 2019.